Composants multiplateformes pour la prise en compte de l'hétérogénéité des terminaux mobiles

Joachim Perchat

Université de Valenciennes et du Hainaut-Cambrésis

Thèse de doctorat

Pour obtenir le grade de Docteur de l'Université de

VALENCIENNES ET DU HAINAUT-CAMBRESIS

Discipline, spécialité selon la liste des spécialités pour lesquelles l'Ecole Doctorale est accréditée :

Sciences Pour l'Ingénieur (SPI) Université Lille Nord-de-France – 072
Spécialité informatique

Présentée par Joachim PERCHAT
à l'université de Valenciennes et du Hainaut-Cambrésis
le 08/01/2015

Ecole doctorale : Sciences Pour l'Ingénieur (SPI)

Equipe de recherche, Laboratoire :

Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines (LAMIH)

Entreprise :

Keyneosoft

Composants multiplateformes pour la prise en compte de l'hétérogénéité des terminaux mobiles

JURY

Président du Jury

— M. Thierry DELOT, Professeur à l'université de Valenciennes

Rapporteurs

— M. Jean Marc PIERSON, Professeur à l'IRIT, Toulouse
— M. Gaël THOMAS, Professeur à Telecom SudParis

Examinateurs

— M. Stéphane FRENOT, Professeur à l'INSA, Lyon

Directeur de thèse : LECOMTE, Sylvain. Professeur. Université de Valenciennes et du Hainaut-Cambrésis.
Co-encadrant de thèse : DESERTOT, Mikaël. Maître de conférence. Université de Valenciennes et du Hainaut-Cambrésis.
Tuteur entreprise : DELCOURT, Nicolas. Directeur Associé. Keyneosoft.

Remerciements

Tout d'abord, je tiens à remercier les membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je tiens à remercier particulièrement Jean Marc Pierson de l'IRIT à Toulouse et Gaël Thomas de Telecom SudParis qui ont accepté d'être les rapporteurs de ma thèse. Je remercie également Stéphane Frenot de l'INSA à Lyon et Thierry Delot de l'université de Valenciennes qui ont accepté d'être examinateur de ma thèse.

Je remercie tout particulièrement mes directeurs de thèses Sylvain Lecomte et Mikaël Desertot pour m'avoir fait confiance pour cette thèse ainsi que pour m'avoir fourni tout leur soutien et leur patience. Je les remercie également pour leurs conseils et leurs remarques qui ont su faire naitre des reflexions profondes en moi. J'espère que nous travaillerons de nouveau ensemble assez rapidement.

Je remercie également Nicolas Delcourt et Alexandre Mayaud pour leur confiance dans ce projet. Sans Keyneosoft, il m'aurait été difficile d'appréhender les enjeux du domaine du mobile et plus généralement de l'informatique ambiante à travers les magasins connectés. Je tiens également à remercier Romain Marion et Jérôme Pramondon qui m'ont apporté leurs connaissances méthodologiques et techniques ces deux dernières années. Je suis heureux de rester à leurs côtés en tant qu'expert technique dans le développement mobile et multiplateforme.

Je remercie l'équipe du LAMIH et plus particulièrement l'équipe DIM dans laquelle j'ai passé la moitié de ces trois années. Je pense notamment à Dana, Olivier et Jonathan qui ont su rendre notre bureau toujours convivial, ainsi que tous les autres, Christophe, Éric, David, Rabbie, Santosh ... Merci pour toutes ces discussions autour de la recherche et de vos domaines respectifs. Elles ont été enrichissantes et m'ont ouvert sur beaucoup de domaines de l'informatique.

Je remercie l'équipe de développement de Keyneosoft, les développeurs mobiles : Laurent, Grégoire, Xavier, Clément ..., les développeurs serveurs et web : Ludovic, Pierrick, Thomas et Nicolas. J'adore nos reflexions autour de la conception de KeyBuild et de nos projets. De plus, vous avez su rendre ces trois années très agréables.

Finalement, je voudrais remercier tous les membres de ma famille et notamment mes parents, mes frères et ma soeur, ma belle famille qui m'ont soutenu pendant ces trois années intenses. Je voudrais remercier tout particulièrement mes grands parents et surtout mon grand père pour s'être intéressé à toutes les étapes de ce travail même si il faut l'avouer le monde du mobile est un monde très lointain pour lui.

Je donne une mention spéciale à ma femme Marie-Aline et notre fils Jordan qui ont toujours été présents moralement et bien plus encore. Je pense surtout au moment où elle a dû ressortir ses talents de développeuse !

Résumé

Ces travaux de thèse visent à diminuer le coût de développement des applications mobiles pour smartphones Android, iOS, etc.

Les applications mobiles sont de plus en plus complexes. Auparavant, une application mobile se contentait d'afficher des données provenant du web. Maintenant, en plus de cela, elles communiquent avec le monde extérieur. Par exemple, certaines applications communiquent avec des montres, avec des écrans de télévision etc. D'autres permettent le scan de codes barres ou encore l'interaction avec des objets réels à travers la réalité augmentée. Les serveurs peuvent envoyer des notifications aux applications, etc. Une application mobile est devenue un logiciel à part entière. Cependant, pour toucher un maximum d'utilisateurs de smartphones, les applications mobiles doivent être conçues, implémentées et déployées sur tous les smartphones possibles. Avec la multiplication des configurations matérielles différentes ainsi que la multiplication des systèmes d'exploitation mobiles, cette tâche devient de plus en plus ardue. En effet, une application mobile doit souvent être réalisée une fois pour chaque plate-forme cible (Android, iOS, Windows Phone 8, etc.). Le temps et le coût de réalisation d'applications mobiles est donc multiplié par le nombre de plates-formes ciblées.

Dans ces travaux, nous proposons de combiner le développement natif avec la programmation par composants. Pour ce faire, nous introduisons la notion de composants multiplateformes. Ce sont des composants qui peuvent être exécutés sur plusieurs plates-formes mobiles. Pour la représentation des composants, nous avons introduit la notion d'interface indépendante à n'importe quelle plate-forme mobile. Ainsi, l'intégration et l'assemblage se font d'une façon unique, que l'on soit dans un environnement de développement Android, iOS ou autre. Pour ce faire, nous avons spécifié un nouveau langage de programmation basé sur les annotations. Cette approche a été validée à travers le développement d'une application mobile pour Android et iOS avec notre solution. L'application implémentée a été réalisée en concordance avec les problématiques que rencontrent les entreprises de développement mobile et plus particulièrement Keyneosoft. Ensuite, nous avons comparé ces versions de l'application avec les versions développées nativement. Nous avons montré qu'avec notre solution nous diminuons le temps de développement d'au moins 30% sans aucune limitation pour les développeurs d'applications (même expérience utilisateur, même performance). Nous avons aussi comparé notre solution avec des produits disponibles sur le marché Phonegap, Titanium mobile et Xamarin. Nous en avons conclu que notre solutiton offrait le plus de possibilités sans aucune limitation.

Mots clés : développement mobile multiplateforme, programmation par composants, composants multiplateformes, assemblage, langage commun, compilateur source-à-source.

Abstract

In this thesis, we aim to decrease the development cost of applications for smartphones running Android, iOS, etc.

Mobile applications are more and more complex. A few year ago, a mobile application was only design to display web content. Today, in addition, they are connected with the external world. For example, some applications are connected with watches, TVs, etc. Mobile applications became real softwares. However, in order to be visible by all smartphones users, mobile applications are designed, developed and deployed on every kind of smartphone. With the increase of the multiplicity of hardware configurations and the diversity of mobile operating systems, this task is becoming more and more laborious. Indeed, a mobile application is often implemented one time for each target platform (Android, iOS, Windows Phone 8, etc.). Therefore, the time and the cost for a mobile application implementation is multiplied by the number of target platforms.

In this thesis, we propose to combine native development with the advantages of component-based software engineering. To do that, we have introduced the concept of multi platform components. Those components are capable to be executed on any mobile platform. In order to describe components, we have introduced interfaces that are independent of any mobile platform. Thus, component integration and assembly are common on Android, iOS and others systems. To achieve that, we have specified a new programming language based on Annotations. We have validated this approach with the implementation of a real mobile application for Android and iOS. We have compared this application with the same application developed natively. Results show that with our solution, developers implement a multi platform application 30% faster than native development. Moreover, our solution does not show any limitation for developers (same user experience, same performances). Finally, we have compared our solution with real products available on the software market: Phonegap, Titanium mobile and Xamarin. This comparison illustrates that our solution provides the best features and does not limit developers possibilities.

keywords: cross-platform mobile development, component-based development, cross-platform components, assembly language, common programming language, source-to-source compiler.

Table des matières

1 Introduction **1**

1.1 Contexte général 1

1.2 Problématique 2

1.3 Cadre de travail 3

1.4 Organisation du mémoire 3

I Problématique et État de l'art **5**

2 Le développement d'applications mobiles pour smartphones **7**

2.1 L'hétérogénéité des terminaux mobiles 7

2.2 La prise en compte de l'hétérogénéité des terminaux mobiles : un coût élevé . 10

2.3 Comment diminuer le coût de développement d'une application mobile multiplateforme ? 12

2.4 Le développement mobile multiplateforme : plusieurs manières d'implémenter la même application 14

2.4.1 MVC : un patron de programmation adapté au mobile 15

2.4.2 Définition des vues 16

2.4.3 Définition des contrôleurs 18

2.4.4 Définition du modèle 19

2.5 Attentes et Éxigences 20

2.6 Conclusion 23

3 Environnement de développement pour applications mobiles multiplateformes **25**

3.1 Introduction 25

3.2 Plusieurs environnements mobiles 26

3.2.1 Android 26

3.2.2 iOS 28

3.2.3 Windows Phone 8 29
3.3 Solutions pour le développement multiplateforme proches des systèmes d'exploitations mobiles 31
3.3.1 Compilateur source à source 31
3.3.2 Applications web 35
3.4 Conclusion 41

4 Modèle de développement prenant en compte l'hétérogénéité des systèmes 43
4.1 Introduction 44
4.2 Les langages spécifiques à un domaine (DSL) et l'ingénierie dirigée par les modèles (IDM) 44
4.2.1 Les DSLs : Domain Specific Language 44
4.2.2 Ingénierie dirigée par les modèles (IDM) 45
4.2.3 Discussions 48
4.3 Environnement d'exécution commun 48
4.3.1 Machine virtuelle 48
4.3.2 Délégation des services sur des serveurs/cloud 50
4.3.3 Portage d'environnements mobiles 51
4.4 La programmation par composants 52
4.5 Conclusion 56

II Contributions 59

5 La programmation par composants pour le développement d'applications mobiles 61
5.1 Introduction 61
5.2 Architecture générale 63
5.2.1 Couche 1 : Structure de l'application 65
5.2.2 Couche 2 : langage universel 66
5.2.3 Couche 3 : Un panel de composants multiplateformes 68
5.3 Compilation et applications générées 70
5.4 Réutilisation 71
5.5 Conclusion 72

6 Définition d'un nouveau genre de composants : des composants multiplateformes 73

6.1 Introduction 74
6.2 Interactions possibles avec un composant multiplateforme 74
6.2.1 Entrée/sortie d'un composant 74
6.2.2 La délégation : un moyen de rendre un composant indépendant de toutes les applications 75
6.2.3 Un exemple concret : "httpRequestManager" 77
6.3 Structure interne d'un composant multiplateforme 79
6.4 Description de la partie cachée d'un composant multiplateforme 80
6.4.1 Une implémentation par plate-forme cible 80
6.4.2 Une interface complémentaire 81
6.4.3 La partie cachée du composant "HttpRequestManager" 81
6.5 Description visible d'un composant multiplateforme 86
6.5.1 Une interface publique 87
6.5.2 La partie visible du composant "HttpRequestManager" 88
6.6 Conclusion 90

7 Langage universel et compilateur 91

7.1 Introduction 91
7.2 Langage commun 92
7.2.1 Besoins 92
7.2.2 Un langage commun basé sur les Annotations 92
7.2.3 Les annotations dans le mobile 93
7.2.4 Les annotations dans notre framework de développement 94
7.2.5 Intégration sous Android et iOS du composant HttpRequestManager avec notre langage commun 97
7.3 Un compilateur source à source flexible et léger 101
7.3.1 Besoins 102
7.3.2 Règles de compilation 102
7.3.3 Génération du code source pour l'appel au composant "HttpRequest-Manager" 105
7.4 Un framework de développement facilement adaptable 108
7.5 Conclusion 110

III Mise en oeuvre et expérimentations 113

8 Mise en oeuvre : COMMON 115

8.1 Introduction . . . 115
8.2 Une base commune . . . 116
8.3 Quelques outils pour faciliter l'utilisation de COMMON . . . 118
8.3.1 Faciliter le développement des composants multiplateformes . . . 118
8.3.2 Faciliter l'intégration des composants multiplateformes . . . 123
8.4 Réalisation du compilateur source à source . . . 128
8.5 Charge de travail . . . 131
8.6 Conclusion . . . 132

9 Évaluation de COMMON 133

9.1 Introduction . . . 133
9.2 Application d'évaluation . . . 134
9.3 Implémentation de LocaPlace avec COMMON . . . 136
9.4 Évaluations . . . 138
9.4.1 Faisabilité de l'application . . . 139
9.4.2 Diminution de la charge de développement . . . 140
9.4.3 Performances . . . 143
9.4.4 Consommation des ressources . . . 147
9.5 Conclusion . . . 151

10 Comparaison de COMMON avec d'autres solutions 153

10.1 Introduction . . . 153
10.2 Présentation des outils . . . 154
10.2.1 PhoneGap . . . 154
10.2.2 Titanium mobile . . . 155
10.2.3 Xamarin . . . 157
10.3 Résultats . . . 158
10.3.1 Expérience utilisateur . . . 159
10.3.2 Performances . . . 162
10.3.3 Utilisation des ressources . . . 165
10.3.4 Discussions . . . 168
10.4 Conclusion . . . 169

IV Conclusions et perspectives **171**

11 Conclusion et Perspectives **173**

11.1 Conclusion 173

11.2 Perspectives 176

Table des figures

2.1 Parts de marché de chaque plate-forme mobile en Europe 8

2.2 Parts de marché de chaque version du système d'exploitation Android 9

2.3 Parts de marché de chaque version du système d'exploitation Windows Phone 10

2.4 Nombre d'applications téléchargées à travers les magasins d'applications Google Play et App Store 13

2.5 Recherche d'une ville avec notre application de test sur Android 14

2.6 Recherche d'une ville avec notre application de test sur iOS 15

2.7 Recherche d'une ville avec notre application de test sur Windows Phone 8 . . 15

2.8 Représentation du design pattern MVC 16

2.9 Exemple de définition d'une vue sur Android 17

2.10 Définition d'une vue sur iOS avec l'interface builder d'XCode 17

2.11 Listener permettant la récupération des interactions utilisateurs sur un champ texte en Java pour Android 19

2.12 Delegate permettant la récupération des interactions utilisateurs sur un champ texte en Objective-C pour iOS 19

2.13 Diagramme de classes : couche modèle de notre application d'autocomplétion 20

2.14 Navigation par pile de vues sur iOS 22

2.15 Navigation par panorama sur Windows Phone 8 23

3.1 Architecture du système d'exploitation Android 28

3.2 Architecture du système d'exploitation iOS 29

3.3 Architecture du système d'exploitation Windows Phone 8 30

3.4 Principe de la compilation source à source application au développement mobile multiplateforme 32

3.5 Principe de l'encapsultation d'un site web dans une application mobile 38

4.1 Principe de l'ingénierie dirigée par les modèles appliquées au développement d'applications mobiles 47

4.2 Principe du développement multiplateforme avec Java ME 49

4.3 Représentation classique d'un composant 52
4.4 Assemblage d'une application à base de composants 53
4.5 Représentation de plusieurs composants implémentés à partir de différents environnements de développement 54
4.6 Remplacement d'une implémentation d'un composant par une autre 56

5.1 Éléments natifs pour le choix d'une date sur iOS et Android 62
5.2 Architecture générale de notre framework de développement permettant la prise en compte de l'hétérogénéité des systèmes d'exploitation mobiles 63
5.3 Différences entre un composant classique et un composant multiplateforme . . 64
5.4 Utilisation de composants classiques à partir d'applications de bureau 67
5.5 Utilisation de composants multiplateformes dans notre solution 69
5.6 Architecture détaillée de notre framework de développement 70
5.7 Application générée par notre compilateur 71
5.8 Réutilisation des composants dans un maximum d'applications 71

6.1 Entrée/sortie d'un composant 75
6.2 Processus synchrone 75
6.3 Processus asynchrone 76
6.4 Interactions entre une application et le composant "ShoppingList" 77
6.5 Entrées/sorties du composant "HttpRequestManager" 78
6.6 Interactions entre une application et le composant "HttpRequestManager" . . 79
6.7 Structure d'un composant multiplateforme dans notre solution 79
6.8 Hiérarchie de l'interface complémentaire d'un composant 82
6.9 Affichage d'un indicateur d'activité dans la barre de statut d'iOS lorsqu'on effectue une connexion internet 84
6.10 Réunion des méthodes natives par fonctionnalités 85
6.11 Hiérarchie de l'interface commune d'un composant 88

7.1 Liens entre l'interface publique d'un composant et nos annotations 97
7.2 Lien entre l'interface publique du composant HttpRequestManager et les annotations permettant de l'intégrer dans une application native 98
7.3 Liens entre l'interface complémentaire d'un composant et le code généré par le compilateur 104
7.4 Transofmration d'une annotation ***@method*** vers un code source swift 109

8.1 Choix des implémentations du composant dont les interfaces doivent être générées 119

8.2 Sélection des interfaces natives publiques du composant HttpRequestManager 120
8.3 Liaisons des différentes implémentations natives d'un composant 122
8.4 Commentaires de la méthode ***cancelHttpRequest()*** 123
8.5 Dossier de distribution du composant "HttpRequestManager" 123
8.6 Résumé des fonctionnalités du composant "HttpRequestManager" 125
8.7 Aide à l'intégration du composant "HttpRequestManager" sur Android 126
8.8 Aide à l'intégration du composant "HttpRequestManager" sur iOS 127
8.9 Description des annotations de notre langage 127
8.10 Processus de pré-compilation pour un projet iOS 129
8.11 Utilisation idéale de notre cross-compilateur 130

9.1 Storyboard représentant l'application LocaPlace 135
9.2 Intégration des composants dans l'application LocaPlace 139
9.3 Interaction de type swipe dans l'application LocaPlace 140
9.4 Temps d'exécution moyen relevé sur plusieurs smartphones android 145
9.5 Temps d'exécution moyen relevé sur plusieurs smartphones iOS 146
9.6 RAM moyenne utilisée après chaque processus sur plusieurs smartphones android 148
9.7 RAM moyenne utilisée après chaque processus sur plusieurs smartphones iOS 149

10.1 Architecture de l'application LocaPlace implémentée avec PhoneGap pour Android et iOS 155
10.2 Architecture de l'application LocaPlace implémentée avec Titanium mobile pour Android et iOS 156
10.3 Architecture de l'application LocaPlace implémentée avec Xamarin pour Android et iOS 158
10.4 Liaisons entre les SDKs natifs et les SDKs de Xamarin 161
10.5 Cinématique de la sélection d'une ville avant de lancer la recherche de lieux dans l'application LocaPlace 163
10.6 Comparaison des temps d'exécution moyens relevés sur un nexus 5 à travers les différentes versions de l'application LocaPlace 164
10.7 Comparaison des temps d'exécution moyens relevés sur un iPhone 5s à travers les différentes versions de l'application LocaPlace 166
10.8 Récupération de la RAM utilisée de l'application LocaPlace implémentée avec COMMON à travers l'application Smart Booster 167

Liste des tableaux

2.1 Contraintes techniques liées aux outils de développement 11
2.2 Contraintes techniques liées au déploiement d'applications mobiles 11

8.1 Différences entre les parsers JavaParser et un parser généré avec ANTLR . . . 117
8.2 Nombre de lignes de code écrites pour réaliser les logiciels liés à COMMON . 131
8.3 Nombre de lignes de code des logiciels utilisés par COMMON 131

9.1 Nombre de lignes de code gagnées avec COMMON 141
9.2 Nombre de lignes de code gagnées avec COMMON sans prendre en compte les spécificités de l'application LocaPlace 142
9.3 Nombre de lignes de code gagnées avec COMMON 142
9.4 Poids de l'application LocaPlace installées sur plusieurs smartphones Android 150
9.5 Poids de l'application LocaPlace lors de son packaging pour être exécutée sur des smartphones iOS 150

10.1 RAM utilisée par chaque version de l'application LocaPlace après une utilisation complète sur un nexus 5 167
10.2 Poids de chaque version de l'application LocaPlace après une installation sur Android et iOS 168
10.3 Comparaison de chaque solution par rapport à nos besoins 170

Liste des Codes

6.1 Interface native pour Android du composant HttpRequestManager 82

6.2 Interface native pour iOS du composant HttpRequestManager 82

6.3 Représentation du delegate **HttpRequestDelegate** en Java pour Android . 83

6.4 Représentation du delegate **HttpRequestDelegate** en Objective-C pour iOS 83

6.5 Interface complémentaire du composant "HttpRequestManager" 85

6.6 Une partie Interface publique du composant "HttpRequestManager" 88

7.1 Signature de l'annotation ***@method*** 94

7.2 Signature de l'annotation ***@var*** 95

7.3 Signature de l'annotation ***@delegate*** 96

7.4 Appel de la méthode ***sendHttpRequest*** avec l'annotation ***@method*** 97

7.5 Annotations permettant de finaliser l'appel au composant "httpRequestManager" 98

7.6 Code du contrôleur de vue dans notre application d'intégration Android . . . 99

7.7 Code du contrôleur de vue dans notre application d'intégration iOS 99

7.8 Code du contrôleur de vue dans notre application d'intégration Android . . . 100

7.9 Intégration d'un appel au "HttpRequestManager" dans un code source natif iOS 101

7.10 Code généré après l'intégration du composant "HttpRequestManager" (Android) 105

7.11 Code généré après l'intégration du composant "HttpRequestManager" (iOS) . 107

8.1 Description du composant HttpRequestManager 120

8.2 Algorithme de compilation d'annotations en code source natif 128

8.3 Commande pour lancer le cross-compilateur 130

9.1 Commande pour calculer le nombre de ligne de code d'un projet android . . . 141

9.2 Commande pour calculer le nombre de ligne de code d'un projet iOS 141

Chapitre 1

Introduction

Dans ce chapitre, nous présentons le contexte général dans lequel ces travaux de thèse ont été effectués. Aujourd'hui, la richesse du domaine du mobile provient notamment du nombre toujours croissant de smartphones et tablettes ainsi que de systèmes d'exploitation disponibles. Cependant, cette richesse est difficile à prendre en compte pour les entreprises de développement mobile et a un coût élevé. Dans cette thèse, nous présentons une nouvelle approche permettant d'atténuer les différences entre mobile pour faciliter le développement d'applications.

1.1 Contexte général

Le domaine du mobile est devenu, depuis le 28 novembre 2007, avec la sortie pour le grand public des smartphones, un domaine de l'informatique à part entière. Le premier smartphone qui a révolutionné ce domaine est l'iPhone d'Apple. Aujourd'hui, la population mondiale est mieux équipée en smartphones et tablettes qu'en ordinateur de bureau. Le cabinet Gartner a publié une étude en 2013 [1] montrant qu'en 2014, le nombre de smartphones et tablettes vendus dans le monde s'élèveront à 2,1 milliards d'appareils contre 300 millions de PC vendus.

Cet attrait rapide pour les smartphones provient notamment de la puissance des appareils qui ne cesse d'augmenter et des nouvelles possibilités qu'ils offrent. Il est maintenant pratiquement possible de tout faire avec son smartphone. Cela n'a été possible que grâce à un nouveau genre d'application mobiles. Ce sont des applications qui sont installées sur le smartphones et qui fonctionnent indépendamment des autres. Leurs points forts sont les possibilités qu'elles ont d'accéder à toutes les fonctionnalités du smartphone et du système d'exploitation installé (GPS, internet, l'accéléromètre, liste des contacts, etc.). Ainsi, de nouveaux usages sont apparus. Par exemple, en mobilité, il est maintenant possible d'acheter son ticket de métro ou de train directement avec son smartphone sans carte bleu. Il est maintenant possible de faire ses courses avec son smartphone. L'utilisateur scanne les produits qu'ils souhaitent acheter au fur et à mesure de ses courses [2]. À la fin, lorsqu'il passe à la caisse, il

1. Étude Gartner sur le nombre de ventes d'appareils informatique entre 2012 et 2014 : http ://www.gartner.com/newsroom/id/2525515

2. Solution de Keyneosoft pour le SelfScanning mobile http ://www.keyneosoft.com/digital-in-store/self-scanning-mobile

lui suffit de valider via une application et de partir avec son caddie. Il est aussi possible de commander un drone à partir d'un smartphone. De plus, ce genre d'appareil ne s'utilise plus uniquement en mobilité mais aussi chez soi, par exemple, pour jouer un jeux vidéo dans son canapé, vérifier ses comptes bancaires, etc. Les applications sont presques devenus infinies.

Pour rendre accessibles toutes ces applications, les fournisseurs de systèmes d'exploitation mobiles se sont dotés de magasins d'applications. C'est à partir de ces magasins que les utilisateurs peuvent rechercher et installer des applications, les mettre à jour, les noter, etc. Ce nouvel écosystème a rendu les applications mobiles pérennes. Aujourd'hui, c'est à travers les applications mobiles que les smartphones sont principalement utilisés.

1.2 Problématique

Plusieurs problématiques liées à l'essor du marché mobile ont vu le jour [Was10] :

— Comment fournir la meilleure expérience utilisateur possible ? L'expérience utilisateur doit être conçue en prenant en compte plusieurs paramètres comme la petite taille des écrans. Nous ne pouvons pas afficher autant de données que sur un ordinateur. Elle peut aussi s'appuyer sur de nouveaux types d'interactions comme le NFC (mettre reference), par exemple. Enfin, le public des applications mobiles est très hétérogène. Il se peut qu'une personne de 50 ans utilise la même application qu'une personne de 20 ans. Il faut alors répondre à leurs besoins respectifs.

— Comment tirer le meilleur parti des smartphones (performances, batteries, sécurité, etc.) ? Le succès d'une application ne dépend pas uniquement des fonctionnalités qu'elle propose mais aussi de ses performances. Elle doit être rapide, ne pas consommer trop de batterie, être sécurisée, etc. La sécurité des données est un point très important pour les smartphones. Il faut s'assurer que les applications installées ne vont pas récupérer toutes les informations de l'utilisateur pour les revendre, par exemple.

— Comment tester et maintenir efficacement une application mobile ? Aujourd'hui, il existe des émulateurs de smartphones. Cependant, lorsque nous testons nos applications, nous nous rendons souvent compte qu'elles ne fonctionnent pas de la même façon. De plus, les tests doivent être effectués sur le plus grand nombre de téléphones possibles (différentes tailles d'écrans, différents processeurs, etc.). La maintenance est aussi difficile. Les systèmes d'exploitation sont souvent mis à jour. Il faut alors re-tester toutes les applications sur la nouvelle version du système d'exploitation.

— Enfin, comment fournir une application mobile pour tous les utilisateurs de smartphones et tablettes du marché ? Aujourd'hui, les smartphones et tablettes sont fabriqués par des constructeurs différents. Le système d'exploitation installé est fourni par des fournisseurs différents. De nos jours, cette hétérogénéité à un coût non négligeable lors du développement d'applications mobiles.

Dans ces travaux de thèse, nous nous attaquons à la dernière problématique. L'objectif est de diminuer le coût de développement des applications mobiles. En effet, avec l'essor du marché des smartphones, beaucoup d'entreprises veulent avoir une visibilité sur les magasin d'applications de chacun des systèmes d'exploitation populaires. Cependant, pour fournir la même application sur tous les magasins d'applications, il faut implémenter une version pour chacun d'eux. Au lieu de développer une seule fois l'application, nous devons la développer 3 ou 4 fois. Cela a pour effet de limiter l'innovation dans le monde du mobile. Nous passons beaucoup trop de temps a porter nos applications sur toutes les plates-formes mobiles au lieu de rechercher de nouveaux usages aux smartphones.

De plus, les applications mobiles sous la forme que nous connaissons maintenant n'en est qu'à ses débuts. Aujourd'hui, ce domaine est un sous-ensemble de l'informatique ambiante [CDST13]. Nous observons de plus en plus de nouveaux systèmes permettant des interactions entre les smartphones et les objets du quotidien. Un exemple concret provient de Samsung qui connecte ses smartphones avec ses télés ou ses montres. Pour arriver à cela, ils ont installé Android sur ces objets dans le but de rendre la communication plus facile. Il faut donc s'attendre à voir les systèmes d'exploitations mobiles, que nous connaissons aujourd'hui sur nos smartphones, à être installés sur une multitude d'appareils différents du quotidien.

1.3 Cadre de travail

Ces travaux de thèse ont été effectués au sein du LAMIH (Laboratoire d'Automatique, de mécanique et d'informatique industrielles et humaines) dans l'équipe Optimob (Optimisation et mobilité) du département informatique en collaboration avec Keyneosoft [3] à travers une thèse CIFRE.

Les 3 objectifs du thème Optimob couvrent les problèmes d'optimisation et décision dans un contexte des systèmes mobiles et embarqués. Ils portent sur : le passage à l'échelle ; l'exploration de nouvelles architectures matérielles et logicielles capables de s'adapter à un changement de contexte ; la conception de méthodes de décision susceptibles de résoudre de manière exacte, ou approchée, des problèmes complexes. Quant à elle, Keyneosoft est une entreprise d'édition de logiciels dans le domaine de la vente. L'objectif est de rendre interactif et cross-canal les points de vente à travers des solutions déployées entre autres sur le web, sur des bornes et sur des appareils mobiles.

Pour Keyneosoft, le fait de diminuer le coût des applications mobiles multiplateformes constitue une avancée considérable pour ses clients. Le développement de telles applications coûte, en effet, très cher. Effectivement, pour la même application, nous devons implémenter autant de fois qu'il y a de plates-formes mobiles ciblées (souvent Android et iOS). Par conséquent, une solution prenant en compte l'hétérogénéité des plates-formes mobiles de façon transparente permettrait à Keyneosoft de se focaliser sur le coeur des applications, c'est-à-dire, sur les fonctionnalités et non sur la réalisation. De plus, avec ce genre de solution, nous pourrions être plus compétitifs en baissant le prix total d'une application multiplateforme.

Le laboratoire est, quant à lui, spécialisé dans le transport. Les applications mobiles étant beaucoup utilisées en mobilité, elles constituent une bonne solution pour réaliser des prototypes et tester des concepts sur un large public et sur des architectures matérielles et logicielles différentes. Comme pour les entreprises, le laboratoire est souvent bloqué par la mise en oeuvre de tels prototypes sur plusieurs plates-formes mobiles. Une solution prenant en compte l'hétérogénéité des plates-formes mobiles permettrait donc de valider ou non des approches plus rapidement et sur un large public.

1.4 Organisation du mémoire

Ce mémoire est divisé en trois parties.

La première partie présente en détail la problématique que nous voulons traiter, chapitre 2. Ensuite, nous passons en revue les solutions qui permettent d'implémenter une application mobile de façon multiplateforme. Ces solutions sont divisées en deux parties, chapitres

3. Site web officiel de keyneosoft : http ://www.keyneosoft.com

3 et 4 : les solutions qui se basent uniquement sur les outils de programmation pour simplifier le développement et les solutions qui structurent l'application de façon à accroitre le développement.

La deuxième partie contient nos contributions. Dans un premier temps, chapitre 5, nous introduisons la programmation par composants pour répondre au problème d'hétérogénéité du mobile. Plus particulièrement, nous combinons le développement natif avec ce type de programmation. Avec notre solution, un composant peut s'intégrer dans n'importe quelle application développée nativement. Ensuite, chapitre 6, nous détaillons la façon dont nos composants sont formés. Nous introduisons l'aspect d'implémentation multiplateforme. Un composant a plusieurs implémentations qui peuvent être exécutées à travers plusieurs environnement d'exécution Android, iOS, etc. Pour autant, il n'expose qu'une seule interface indépendante de toutes les implémentations et donc de toutes les plates-formes cibles. Enfin, chapitre 7, nous présentons un langage de programmation commun à n'importe quelle plateforme mobile permettant d'intégrer nos composants. Ce langage est suffisamment flexible pour s'intégrer dans n'importe quel code source natif. De plus, le code écrit avec ce langage est le même sur Android, iOS, etc. Pour ce faire, nous nous basons sur l'interface unique des composants.

Enfin, dans la troisième partie, nous expliquons la mise en oeuvre de notre solution à travers un framework nommé COMMON pour "Component Oriented programming for Mobile Multi Os iNtegration", chapitre 8. Nous évaluons ensuite cette solution en la comparant au développement natif, chapitre 9. Nous terminons cette partie, chapitre 10, avec une comparaison entre COMMON et d'autres outils matures disponibles sur le marché.

Pour finir, chapitre 11, nous concluons la thèse et proposons quelques perspectives de recherche soulevées par ce travail.

Première partie

Problématique et État de l'art

Chapitre 2

Le développement d'applications mobiles pour smartphones

Aujourd'hui, les richesses du domaine du mobile sont son hétérogénéité et son évolution rapide. Cependant, ces richesses ne permettent pas aux entreprises de développer, distribuer et maintenir leurs applications multiplateformes de façon simple et rapide. Dans ce chapitre, nous donnons les différentes contraintes techniques liées aux développement de telles applications. Nous voulons identifier les verrous auxquels se heurtent les entreprises de développement mobiles. Nous terminons par la liste des besoins et exigences que nous attendons d'un outil permettant de simplifier le développement d'applications mobiles multiplateformes.

Sommaire

2.1 L'hétérogénéité des terminaux mobiles 7
2.2 La prise en compte de l'hétérogénéité des terminaux mobiles : un coût élevé 10
2.3 Comment diminuer le coût de développement d'une application mobile multiplateforme ? 12
2.4 Le développement mobile multiplateforme : plusieurs manières d'implémenter la même application 14
2.4.1 MVC : un patron de programmation adapté au mobile 15
2.4.2 Définition des vues 16
2.4.3 Définition des contrôleurs 18
2.4.4 Définition du modèle 19
2.5 Attentes et Éxigences 20
2.6 Conclusion 23

2.1 L'hétérogénéité des terminaux mobiles

Les entreprises de développement d'applications mobiles se heurtent à deux freins majeurs dans le déploiement de leurs applications. Le premier est l'hétérogénéité des architec-

tures matérielles sur lesquelles leurs applications peuvent être exécutées. Le deuxième est l'hétérogénéité des systèmes d'exploitation installés sur les smartphones.

Les constructeurs de smartphone, parmi lesquels les plus populaires sont Samsung, Apple Huawei et LG [1], fournissent des smartphones ayant tous des architectures matérielles différentes. Ainsi, les smartphones ont tous des processeurs différents, des écrans différents (résolution, taille), des caméras différentes (résolution) ... Même si il existe beaucoup de différences matérielles, celles-ci sont en partie cachées par le système d'exploitation installé sur le smartphone. Par exemple, Android est installé sur une multitude d'appareils. Les entreprises doivent donc uniquement implémenter leurs applications pour Android et non pour un smartphone en particulier. Malgré cela, toutes les différences matérielles doivent être prises en compte lors de la conception et le développement d'une application. Il faut toujours se souvenir qu'une application mobile peut être exécutée sur des smartphones plus ou moins performants. Ainsi, une application permettant de faire de la reconnaissance optique de caractères en temps réel ne fonctionnera pas si la caméra du smartphone sur lequel elle s'exécute a une résolution trop faible. De la même manière, une application ne doit pas lancer un traitement de plusieurs centaines de milliers d'itérations sur un smartphone ayant un processeur trop lent. L'objectif est de toujours fournir la meilleure expérience utilisateur possible en fonction du support sur lequel l'application fonctionne. Pour arriver à cela, nous devons faire des compromis entre l'utilisation des ressources et les applications développées.

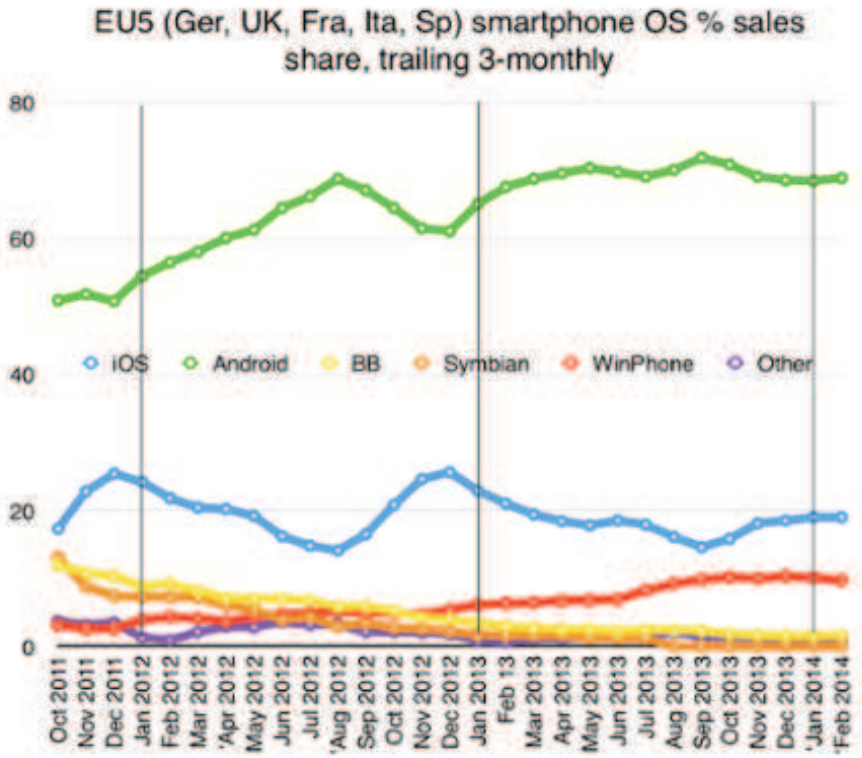


FIGURE 2.1 – Parts de marché de chaque plate-forme mobile en Europe

En plus de l'hétérogénéité engendrée par la multitude de constructeurs de smartphones, il existe plusieurs fournisseurs de système d'exploitation mobile : android de Google, iOS d'Apple, Windows Phone de Microsoft, Symbian OS de Nokia, BlackBerry OS de RIM, etc.

1. Vente mondiale de smartphones en 2013 : http ://www.gartner.com/newsroom/id/2665715

Sur la figure 2.1 [2], nous donnons les parts de marché de chacune de ces plates-formes entre 2011 et 2014. En 2011, au début de cette thèse, les systèmes d'exploitation dominants sur le marché européen était android (50%), iOS (18%), Symbian (14%) et Blackberry (13%). Aujourd'hui en 2014, les parts de marché ont beaucoup évolué. Les systèmes d'exploitations dominants sont maintenant android (70%), iOS (19%) et windows phone (11%).

Enfin, il se peut qu'il y ait, pour un système d'exploitation donné, une disparité entre les versions de celui-ci installées sur des smartphones. Android se trouve dans ce cas de figure. Sur la figure 2.2 [3], nous pouvons voir qu'il y a encore 14,9% des smartphones Android sous la version 2.3, 29% sous la version 4.1 et seulement 13,6% sous la version 4.4. Cela s'explique, d'une part, par le fait que les utilisateurs Android ne mettent pas à jour leurs smartphones et, d'autre part, parce que certains smartphones ayant de trop petites configurations matérielles ne peuvent pas faire fonctionner les nouvelles versions du système d'exploitation. Par exemple, pour la version 4.4, il faut que le smartphone ait au minimum 512MB de RAM disponible pour que la version fonctionne. Cette disparité oblige donc les entreprises de développement d'applications Android à fixer une version minimum sur laquelle leurs applications peuvent être exécutées. Elles doivent choisir entre fournir une application pour le dernier système d'exploitation sorti. Ainsi, elles pourront utiliser toutes les dernières technologies fournies par cette version. Ou, si elles veulent toucher un maximum d'utilisateurs, elles se limiteront à fournir une application moins moderne pour la version 2.3 d'Android. Microsoft a eu le même problème avec Windows Phone. Windows Phone 8 a mis deux ans pour s'imposer face à la version antérieure Windows Phone 7. Aujourd'hui, figure 2.3 [4], Windows Phone 7 représente moins de 20% de parts de marché. Sur iOS, cette disparité est moindre. Les derniers chiffres récoltés par Apple montre qu'iOS 7, l'avant dernière version du système d'exploitation, est installée sur 89% des smartphones Apple [5]. La mise en place de cette version sur 89% des appareils a été effectuée en quelques mois.

Version	Codename	API	Distribution
2.2	Froyo	8	0.8%
2.3.3 - 2.3.7	Gingerbread	10	14.9%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	12.3%
4.1.x	Jelly Bean	16	29.0%
4.2.x		17	19.1%
4.3		18	10.3%
4.4	KitKat	19	13.6%

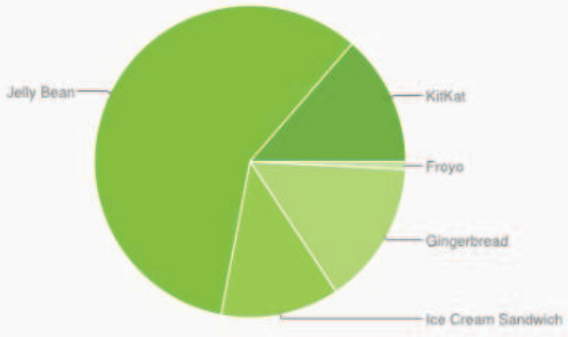


FIGURE 2.2 – Parts de marché de chaque version du système d'exploitation Android

Pour récapituler, si une entreprise de développement d'applications mobiles veut toucher le maximum d'utilisateurs en déployant ses applications sur les magasins d'applications officiels

2. Part de marchés des différentes plates-formes mobiles entre 2011 et 2014 : http ://www.theguardian.com/technology/2014/apr/01/moto-g-boosts-motorola-mobile-smartphone-sales

3. Parts de marché pour chaque version du système d'exploitation Android en juin 2014 : https ://developer.android.com/about/dashboards/index.html Il est possible que le lien ne fournisse pas la même figure que celle présentée ici, la page web est souvent mise à jour par Google

4. Parts de marché pour chaque version du système d'exploitation Windows Phone en avril 2014 : http ://blog.adduplex.com/2014/04/adduplex-windows-phone-statistics.html

5. Keynote Apple à WWDC (Apple Worldwide Developers Conference) 2014 : https ://itunes.apple.com/fr/podcast/apple-keynotes-1080p/id509310064 ?mt=2 (cf : vidéo à la quarante septième minute)

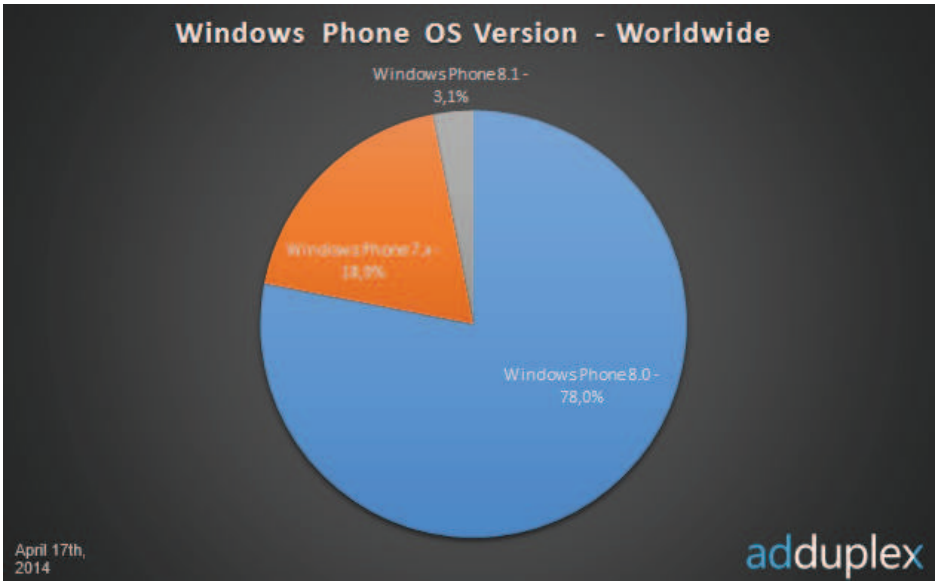


FIGURE 2.3 – Parts de marché de chaque version du système d'exploitation Windows Phone

de chacun des acteurs dominant du marché, elle devra implémenter son application, à la fois, pour iOS 7, Android 2.3 et Windows Phone 8. Pour ce faire, elle devra implémenter trois versions de son application avec des contraintes techniques différentes pour chacune d'entre elles. Par la suite, nous identifions quelques unes de ces contraintes.

2.2 La prise en compte de l'hétérogénéité des terminaux mobiles : un coût élevé

Les tableaux 2.1 et 2.2 résument quelques contraintes techniques à respecter pour développer une application pour Android, iOS et Windows Phone 8. Tout d'abord, nous constatons qu'une entreprise doit avoir des développeurs ayant des connaissances dans trois langages différents : le Java pour Android, l'Objective-C pour iOS et le C# pour Windows Phone 8. De plus, chaque version de l'application devra être implémentée avec des environnements de développement différents : eclipse [6] pour Android, XCode [7] pour iOS et Visual Studio [8] pour Windows Phone. Dans le même ordre d'idée, l'entreprise devra avoir plusieurs types d'ordinateurs : un Mac avec Mac OS X installé et un PC avec Windows 8 pour utiliser respectivement le SDK [9] iOS et le SDK Windows Phone 8. Le SDK d'Android peut être utilisé sur un Mac ou un PC. Ensuite, pour certaines plates-formes, l'utilisation du SDK, à des fins commerciales, peut être payante. C'est le cas d'iOS, il faut avoir un compte développeur chez Apple qui coûte à l'entreprise 99$ par an. Sur Android et Windows Phone 8, ce genre de compte est uniquement nécessaire pour la publication des applications.

Après avoir développé leurs applications en suivant les contraintes fournies par les environnements de développement respectifs de ces systèmes d'exploitation, les développeurs doivent tester leurs applications, tableau 2.2. Pour cela, ils devront avoir à leur disposition des smartphones ayant des architectures matérielles différentes pour chaque plate-forme cible. Par exemple, sur Android ils devront tester leurs applications sur des smartphones dernière

6. Site officiel d'eclipse : http ://www.eclipse.org
7. Site officiel d'XCode : https ://developer.apple.com/xcode/
8. Site officiel de Visual Studio : http ://www.microsoft.com/france/visual-studio/
9. SDK : Software Development Kit

Table 2.1 – Contraintes techniques liées aux outils de développement

plate-forme mobile	Environnement de développement	Langage de programmation	Développement sur	Débugueur
Android	Eclipse	Java	Multiplateforme	DDMS et LogCat
iOS	XCode	Objective-C	Mac OS X	Intruments
Windows Phone 8	Visual Studio	C# et VB.Net	Windows 8 version 64 bits	Intégré dans Visual Studio

Table 2.2 – Contraintes techniques liées au déploiement d'applications mobiles

plate-forme mobile	Distribution d'applications	Validation de l'application	Machine virtuelle	Compte développeur	Parc de smartphones
Android	Google play ou autres	Pas de validation	Dalvik VM	Compte pour la publication d'application Google Play 25$	Hétérogène : provenant de différents constructeurs
iOS	App store	Certification par des opérateurs d'Apple	Interdit	iOS developer program 99$ par an	Homogène : parc géré par Apple
Windows Phone 8	Windows Phone store	Certification par des opérateurs Microsoft	CLR	Compte développeur 99$ par an	Hétérogène : provenant de différents constructeurs avec une configuration minimale à respecter

génération et sur des smartphones plus anciens ou ayant de moins bonnes capacités. Le risque, en ne testant pas leurs applications sur plusieurs smartphones, est d'avoir des comportements différents en fonction de l'architecture sur laquelle leurs applications sont déployées et par la suite avoir des mauvais commentaires et des mauvaises notes sur les magasins d'applications sur lesquels leurs applications sont disponibles. Cette phase est donc primordiale.

Après avoir implémenté et testé leurs applications sur tous les smartphones en leur possession, les entreprises de développement d'applications mobiles peuvent enfin soumettre leurs applications sur les magasins d'applications de chacun des systèmes d'exploitation : Google Play pour Android, App Store pour iOS, Windows Phone store pour Windows Phone 8. Il est indispensable de passer par les outils de chaque plate-forme pour déployer son application. C'est le seul moyen de rentabiliser son application. De plus, pour certaines plates-formes, c'est l'unique moyen de distribuer son application. Sur iOS, il n'est pas possible de partager une application en passant par un autre système que l'App Store. Sur Android, cette pratique est possible. Cependant les utilisateurs ont pris l'habitude de rechercher des applications sur

les magasins d'applications officiels. La publication des applications sur ces magasins est soumise à des règles très strictes [CZHNI+14]. Il y a peu de développeurs d'applications mobiles qui réussissent à publier leurs applications sur l'App Store au premier essai. Chaque marque propriétaire (iOS et Windows Phone) protège au maximum ses intérêts. Par exemple, à partir d'iOS 4, Apple a refusé toutes les applications qui permettaient la détection de code barre à partir de photos. Leurs objectifs sont de fournir la meilleur expérience utilisateur possible pour leur plate-forme. La nouvelle règle est de détecter les codes barres directement à partir du flux de la caméra. Sur Windows Phone 8, il y a aussi une phase de certification avant l'acceptation finale de l'application. Pendant cette phase, les performances de l'application seront testées (temps de démarrage de l'application, mémoire utilisée, poids de l'application ...) ainsi que les bonnes pratiques introduites par Modern UI.

Après avoir déployé leurs applications sur les magasins d'applications officiels de chaque plate-forme, les entreprises doivent assurer une maintenance accrue de celles-ci. En effet, les systèmes d'exploitation mobiles ont tendance à être souvent mis à jour avec parfois des modifications majeures dans le comportement des applications. Cela a notamment été le cas lors du passage d'iOS 6 à iOS 7. Cette nouvelle version a engendré de nombreux changements dans nos applications au niveau du graphisme. Ce problème arrive aussi sur Android et Windows Phone. Il y a eu, par exemple, un changement du système de cartographie sur ces deux plates-formes qui nous a obligé à mettre à jour nos applications utilisant ce type d'éléments. La phase de maintenance est d'autant plus contraignante qu'il y a de systèmes d'exploitation cibles. Nous devons vérifier, à chaque montée de versions des systèmes cibles, que nos applications sont encore compatibles avec ces nouveaux environnements.

Pour récapituler, le développement d'applications mobiles coûte très cher. Une entreprise de développement d'applications mobiles doit fournir des formations à ses développeurs pour se perfectionner sur chacune des plates-formes cibles de ses applications. Après ces formations, elle doit acheter beaucoup de matériels que ce soit pour commencer le développement (achat de MAC et de PC) ou pour tester ses applications (plusieurs smartphones ayant des architectures matérielles très différentes par plate-forme). L'objectif de cette thèse est donc de diminuer le coût de développement d'une application mobile. Une solution est d'atténuer certaines contraintes liées à l'hétérogénéité des terminaux mobiles que ce soit au niveau architecture matérielle mais surtout au niveau architecture logicielle : les systèmes d'exploitation mobiles. Pour cela, nous devons identifier parmi toutes les contraintes exposées ci-dessus lesquelles nous pouvons atténuer.

2.3 Comment diminuer le coût de développement d'une application mobile multiplateforme ?

Sur les tableaux 2.1 et 2.2, nous avons différencié deux niveaux de contraintes pour développer et déployer une application mobile sur plusieurs plates-formes :

- Les outils de développement utilisés pour implémenter une application pour chaque plate-forme, tableau 2.1.
- Les contraintes liées au déploiement d'une application mobile sur les magasins d'applications officiels de chaque plate-forme, tableau 2.2.

Dans cette section, nous allons reprendre ces deux points et ajouter les contraintes techniques liées au développement des applications pour en déduire sur quelle phase influer pour diminuer le coût de développement d'une application mobile multiplateforme.

Toutes les règles liées au déploiement d'une application mobile sur les magasins d'applications officiels de chacun des système d'exploitation cibles sont fixées par les fournisseurs

de ces plates-formes et sont très strictes. Le risque, en ne les respectant pas, est de se voir refuser le déploiement de nos applications. Aujourd'hui, ces magasins ont fait leurs preuves comme nous le montrons sur la figure 2.4 [10]. En 2013, sur Google Play, il y a eu plus de 48 milliards de téléchargements d'applications Android. Au même moment, sur l'App Store, il y avait plus de 50 milliards d'applications iOS téléchargées. Vu ces chiffres, il est indispensable pour une entreprise de développement d'applications mobiles d'avoir une visibilité sur ces magasins d'applications. C'est le seul moyen pour toucher un maximum d'utilisateurs de smartphones. De toute façon, comme nous l'avons signalé dans la section précédente, sur iOS il est impossible de déployer une application pour le grand public autrement. Il est donc obligatoire de respecter les règles au maximum pour avoir une visibilité sur ces magasins.

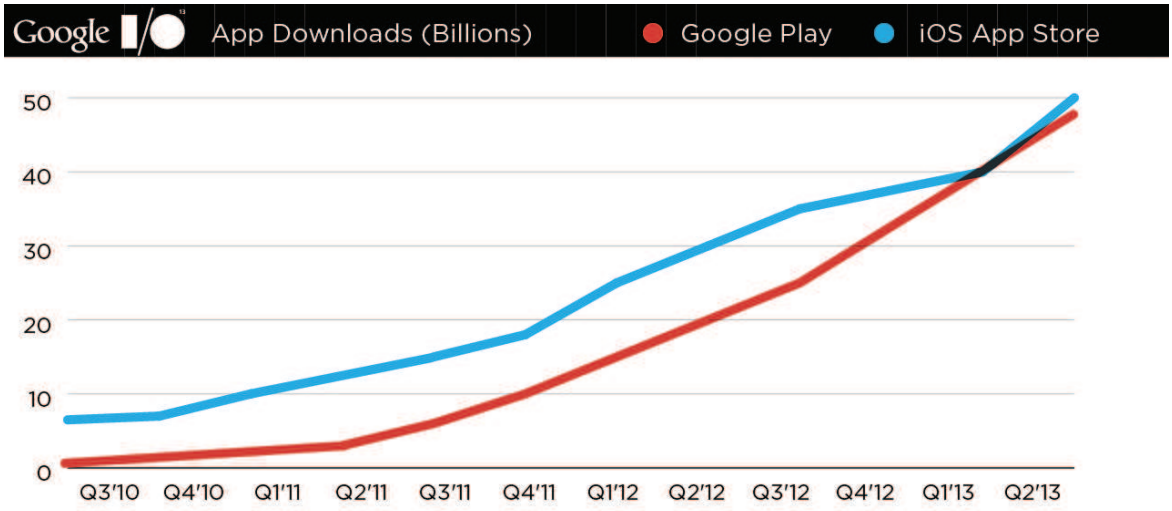


FIGURE 2.4 – Nombre d'applications téléchargées à travers les magasins d'applications Google Play et App Store

Nous ne voulons pas non plus nous passer des outils de développement utilisés pour l'implémentation de nos applications pour chacune des plates-formes. En effet, aujourd'hui, ces outils sont très performants. En plus de nous permettre l'implémentation d'applications mobiles, ils nous permettent de débogguer efficacement nos applications. Par exemple, Google fournit, par l'intermédiaire d'eclipse, des moyens pour monitorer l'utilisation des ressources d'un smartphone (utilisation de la RAM ...). Apple nous fournit le même type d'outil avec le profiler d'Instruments. Cet outil nous permet d'avoir en temps réel l'état de tous les éléments utilisés à un moment précis dans l'application. De plus, en remplaçant les outils de développement officiels de chaque plate-forme en un seul IDE [11] commun, le risque serait de passer notre temps à mettre à jour notre IDE pour prendre en compte les mises à jour des outils officiels. En effet, nous ne pourrions pas réellement remplacer les outils existants vu qu'il est obligatoire de les utiliser pour pouvoir déployer nos applications sur les magasins d'applications officiels. Un IDE commun ne ferait alors qu'utiliser les outils officiels de chacune des plates-formes pour générer des applications.

Il nous reste, alors, les contraintes techniques liées à la phase de développement d'applications mobiles. C'est pour nous, sur cette phase, qu'il faut influer si nous voulons diminuer le coût de développement d'une application mobile. Aujourd'hui, nous ne sommes pas contraint sur la façon de développer une application mobile. Nous verrons dans le chapitre 3 qu'il est même possible, par exemple, d'écrire son application à partir d'un seul code commun et

10. Évènement Google I/0 : https ://developers.google.com/events/io/
11. IDE : Integrated Development Environment

ensuite de compiler ce code source pour plusieurs SDK différents. Dans ce cas, le gain est considérable. Au lieu de développer deux ou trois fois la même application, nous l'implémentons une seule fois.

Par la suite, nous identifions, à travers le développement d'une application simple, plusieurs techniques de développement différentes utilisées par chaque plate-forme. Si nous arrivons à atténuer ces techniques de développement différentes, nous arriverons automatiquement à baisser le coût de développement d'une application mobile multiplateforme.

2.4 Le développement mobile multiplateforme : plusieurs manières d'implémenter la même application

Nous avons implémenté une application basique sous android, iOS et Windows Phone 8. L'objectif est d'identifier toutes les différences possibles lors du développement de la même application pour ces trois plates-formes. Plusieurs comparaisons similaires ont déjà été effectuées [GR11, Här13, BFV13, GHGY14]. L'application implémentée permet de rechercher une ville française en fonction de son nom. C'est une fonctionnalité que nous intégrons souvent dans nos applications mobiles (inscription à un forum, recherche de points d'intérêts, etc.). Dans notre exemple, l'application a un seul écran contenant un champ texte et une liste. Lorsque l'utilisateur tape des caractères dans le champ texte, l'application autocomplète le texte déjà tapé et propose à l'utilisateur le noms des villes commençant par ce texte. Pour récupérer les données, nous utilisons une base de données SQLite. Sur les figures 2.5, 2.6, 2.7, nous récapitulons les interactions possibles sur chacune des plates-formes avec un storyboard. À première vue, cette application se comporte de la même façon que l'on se trouve sur android, iOS ou Windows Phone 8. Malgré cela, le développement est très différent.

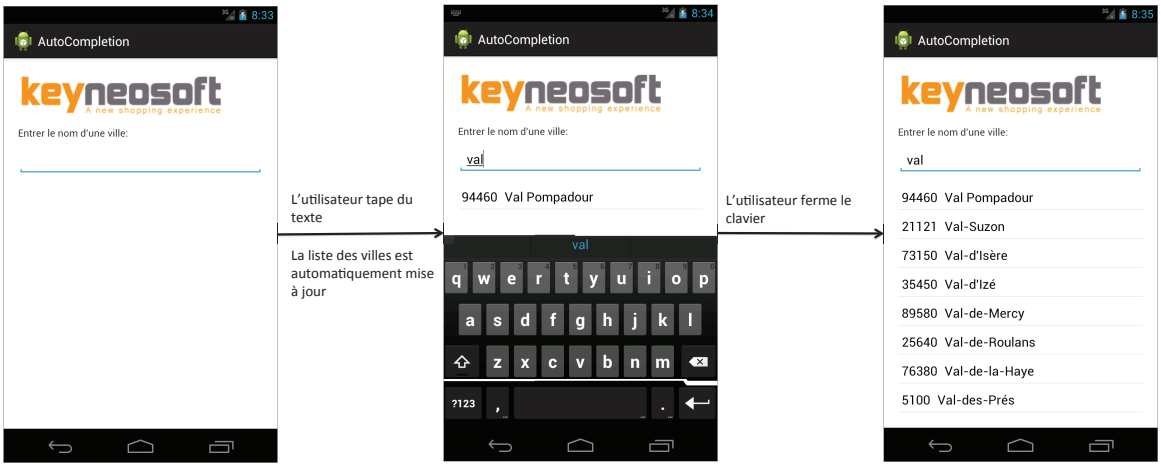


FIGURE 2.5 – Recherche d'une ville avec notre application de test sur Android

Tout d'abord, pour implémenter chaque version, nous utilisons un environnement de développement différent. Nous avons utilisé Eclipse pour Android, XCode pour iOS et Visual Studio pour Windows Phone 8. Malgré une utilisation très différente de ces environnements, comme nous l'avons souligné dans la section précédente, nous voulons garder ces outils. Il faut cependant se rendre compte qu'il est difficile pour un développeur de passer d'un environnement de développement à un autre. Il y a toujours un temps d'adaptation.

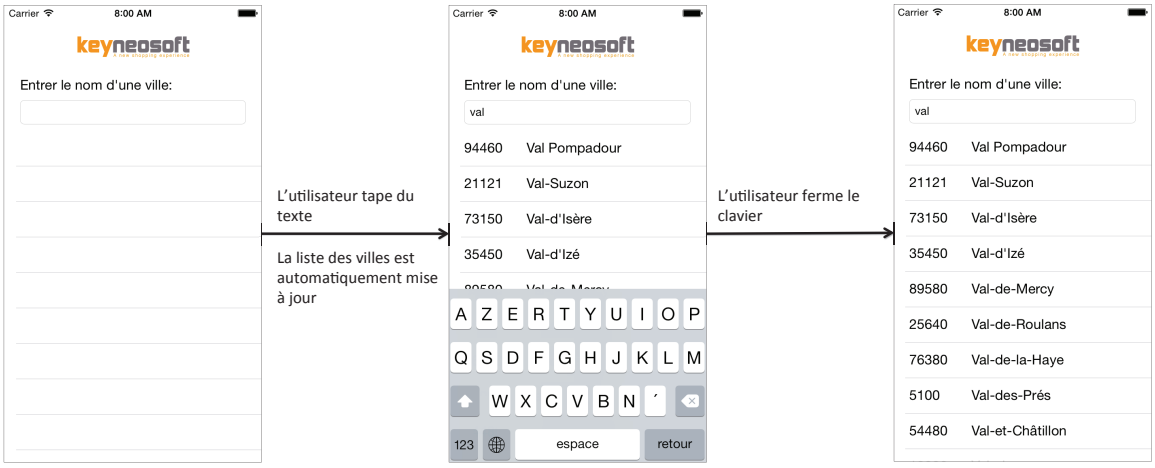


FIGURE 2.6 – Recherche d'une ville avec notre application de test sur iOS

FIGURE 2.7 – Recherche d'une ville avec notre application de test sur Windows Phone 8

2.4.1 MVC : un patron de programmation adapté au mobile

Avant de continuer, il est important de noter que plusieurs plates-formes mobiles nous permettent et nous encouragent à utiliser une architecture MVC (MVC pour Modèle, Vue, Contrôleur) [KP88] pour structurer nos applications. Le patron de conception MVC permet aux développeurs de diviser leurs codes sources en trois couches distinctes :

- Modèle : cette couche représente le coeur de l'application. Dans une application mobile, elle consiste souvent à récupérer les données à partir d'une base de données ou d'un service web et de les traiter en fonction de l'affichage souhaité par les utilisateurs.
- Vue : dans une application mobile, les vues permettent d'afficher les données récupérées à partir du modèle à l'utilisateur. Elles permettent aussi de capter les interactions utilisateurs sur les éléments graphiques de l'interface (clic, swipe ...). Après une interaction utilisateur, les vues transmettent les évènements détectés aux contrôleurs.
- Contrôleur : les contrôleurs s'occupent de faire le lien entre les vues et le modèle. Ils récupèrent les évènements utilisateurs et en fonction de l'état dans lequel se trouve l'application, effectuent les actions souhaitées (changement de vue ...). Si ces actions nécessitent un changement de la couche modèle, le contrôleur s'occupe de demander ces changements. Ensuite, il notifie la vue concernée pour qu'elle mette à jour son état et affiche les nouvelles données.

Sur la figure 2.8, nous présentons les différentes interactions possibles entre ces trois couches. Dans la suite, nous analysons de façon séparée ces trois couches à travers notre application mobile.

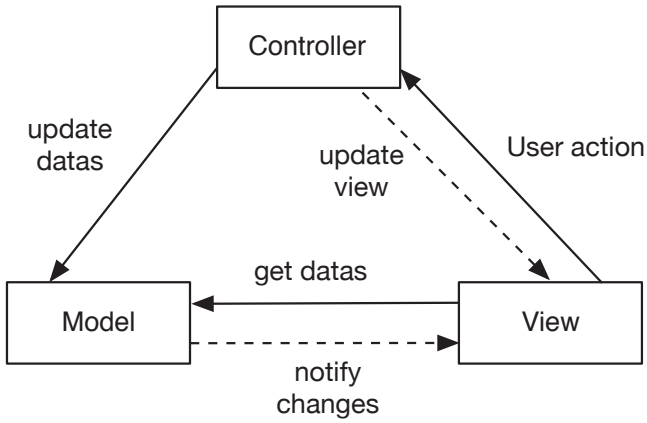


FIGURE 2.8 – Représentation du design pattern MVC

2.4.2 Définition des vues

L'interface utilisateur est implémentée de façon différente pour chaque plate-forme.

Sur Android, nous devons implémenter des fichiers XML. Chaque fichier XML pouvant correspondre à une vue. À l'intérieur de ce fichier, nous décrivons, via des balises XML, le placement de chacun des éléments de l'interface, voir figure 2.9. Parmi ces éléments, nous différencions les conteneurs des éléments qui seront affichés. Un conteneur peut contenir soit des conteneurs soit des éléments graphiques (boutons, champs texte, etc.). Chaque conteneur permet d'afficher des éléments graphiques en fonction d'une stratégie précise (les uns à la suite des autres verticalement ou horizontalement, etc.). Dans le cas d'Android, la taille des éléments graphiques n'est pas fixé dés le départ mais est calculé à l'exécution en fonction de la taille du smartphone sur lequel l'application fonctionne. Sur la figure 2.9, nous ne faisons pas allusion à une taille en pixel par exemple. Ce système est indispensable car une application Android peut fonctionner sur plusieurs smartphones ayant des tailles différentes d'écrans. Il n'est donc pas conseillé de fixer numériquement la taille d'un élément. Par exemple, si nous choisissons de fixer la taille d'un élément à 500 pixels en largeur alors que le smartphone sur lequel l'application s'exécute n'a q'un écran ayant une largeur de 300 pixels, alors il y aura un problème d'affichage. Une partie de l'élément graphique ne sera pas affiché sur l'écran de l'utilisateur.

Sur iOS, nous utilisons l'outil "Interface Builder", figure 2.10. Cet éditeur graphique nous permet de construire nos vues de façon graphique. Pour ce faire, nous choisissons les éléments graphiques à afficher dans une palette d'éléments graphiques, puis avec une action de drag and drop nous déplaçons l'élément sur notre vue. Sur iOS, nous n'avons pas les mêmes problèmes de taille que sur Android. Aujourd'hui, il n'existe que 4 tailles d'écrans différentes sur iOS. L'éditeur nous permet donc d'entrer des tailles fixes pour chacun de nos éléments et de visualiser directement le résultat. Nous avons quand même la possibilité de choisir de façon graphique une stratégie de redimensionnement des éléments lorsque l'on passe d'un iPhone 4 ou 4s (3,5 pouces) vers un iPhone 5, 5C ou 5S (4 pouces) ou encore sur un iPhone 6 (4,7 pouces) ou 6 Plus (5,5 pouces).

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ImageView
        android:id="@+id/logo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="10dp"
        android:src="@drawable/logo_keyneosoft" />

    <TextView
        android:id="@+id/cityLabel"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/logo"
        android:layout_margin="5dp"
        android:text="@string/cityLabel" />

    <EditText
        android:id="@+id/cityNameEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/cityLabel"
        android:layout_margin="5dp" />

    <ListView
        android:id="@+id/cityListView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_below="@id/cityNameEditText"
        android:layout_margin="5dp" />

</RelativeLayout>
```

FIGURE 2.9 – Exemple de définition d'une vue sur Android

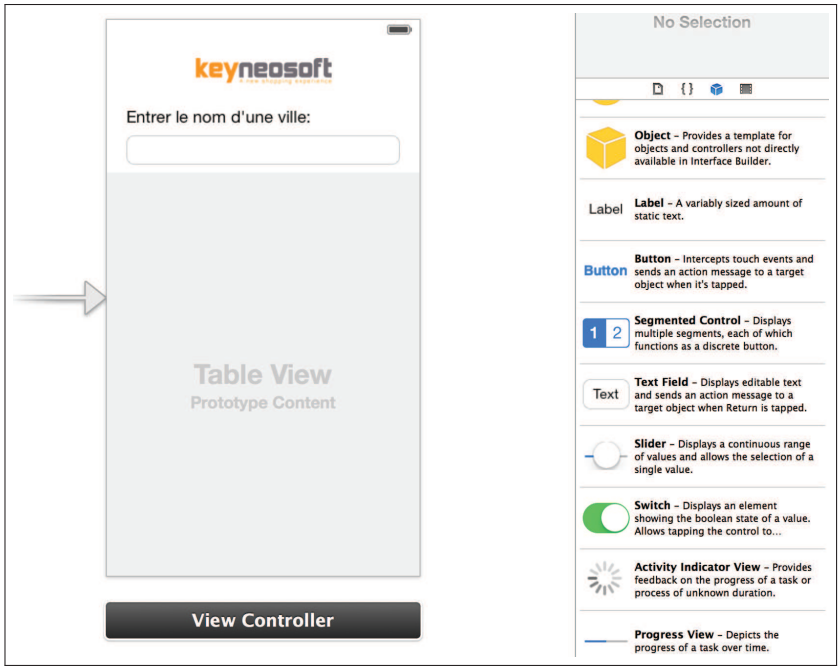


FIGURE 2.10 – Définition d'une vue sur iOS avec l'interface builder d'XCode

Sur windows phone 8, nous devons implémenter l'interface à partir de fichiers XAML. Nous décrivons comme sur Android les vues en XML. Chaque balise correspond à un conteneur ou à un élément graphique. Pour les mêmes raisons que sur Android, nous ne fixons pas la taille des éléments car l'application peut fonctionner sur des écrans ayant des tailles et résolutions différentes.

Avec cet exemple simple, nous observons que la constitution de l'interface graphique pour chacun des systèmes d'exploitations est très différente. Ces différences ne sont pas dues à des utilisations différentes des environnements de développement mais bien à des conceptions et techniques de développement complètement différentes des SDK de chacune des plates-formes cibles. Sur Android et Windows Phone 8, une application peut fonctionner sur une palette très large de téléphone ayant des écrans différents. Le SDK nous fournit donc une manière dynamique de fixer les tailles de nos éléments graphiques. Il nous fournit aussi un système de conteneurs qui n'existe pas sur iOS. Sur iOS, ce problème n'existe pas. Apple gère son propre parc de smartphones avec des petites tailles d'écrans à prendre en considération.

2.4.3 Définition des contrôleurs

Le lien entre la partie interface graphique et le code source de l'application passe par un contrôleur. Pour rappel, c'est le contrôleur qui gère le comportement de l'application lors d'une action utilisateur. Prenons par exemple, l'action "tapper un caractère dans un champ texte".

Sur Android, la récupération des interactions utilisateurs sur nos composants graphiques se fait à travers des "listeners". C'est un patron de programmation très connu dans la programmation orientée objet avec Java. Nous associons un listener à notre objet et dans chacune des méthodes du listener, nous implémentons l'action à effectuer en fonction de l'état de l'application. Sur la figure 2.11, nous avons, par exemple, implémenté le listener **TextWatcher** permettant de récupérer les caractères tapés sur notre champ texte au fur et à mesure de l'exécution de l'application. Sur iOS, ce procédé n'existe pas. En Objective-C, nous devons passer par un système de délégation. Dans notre cas, nous devons implémenter le delegate **UITextFieldDelegate**. Contrairement à Android où le lien entre un listener et un élément graphique est effectué dans le code source, sur iOS ce lien est fait de façon graphique dans "Interface builder". L'implémentation se fait ensuite dans le code source du contrôleur, figure 2.12. Sur Windows Phone 8, la liaison entre les éléments graphiques et les évènements utilisateurs se font comme ce que l'on connait en HTML. En effet, dans la description XML de l'élément graphique, il est possible de lier à des évènements, des méthodes programmées en C# dans un autre fichier.

Le lien entre une vue et un contrôleur se fait encore de manière différente sur toutes les plates-formes cibles de notre application. La différence est significative pour la gestion des interactions utilisateurs : listener pour Android, delegate pour iOS, association d'actions à des méthodes pour Windows Phone 8. Même si l'on pourrait croire que ces procédés sont les mêmes, un listener permet de recevoir une notification d'un évènement alors qu'un delegate permet de recevoir la notification d'un évènement et oblige le contrôleur à traiter l'évènement. Par exemple, sur la figure 2.12, nous devons, en plus de recevoir l'information sur le changement de texte (méthode : ***textField :shouldChangeCharactersInRange :replacementString :***), retourner un booléen permettant de notifier au système si le caractère entré par l'utilisateur peut être ajouté au texte contenu dans le champ texte ou non. Le système iOS nous délègue certains traitements, ce qui n'est pas du tout le cas sur Android. Cette différence majeure oblige les développeurs à changer complètement leur façon de penser lorsqu'ils changent de versions d'applications.

```java
cityEditText.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start,
            int count, int after) {
    }
    @Override
    public void onTextChanged(CharSequence s, int start,
            int before, int count) {

    }
    @Override
    public void afterTextChanged(Editable s) {
        CityDAO cityDAO = new CityDAO(PlaceholderFragment.this
                .getActivity().getApplicationContext());
        cityList = cityDAO.getCityList(s.toString());

        cityListAdapter.setCityList(cityList);
        cityListAdapter.notifyDataSetChanged();
    }
});
```

FIGURE 2.11 – Listener permettant la récupération des interactions utilisateurs sur un champ texte en Java pour Android

```objc
//
//  ViewController.h
//  AutoCompletionThese
//
//  Created by Joachim on 09/05/2014.
//  Copyright (c) 2014 Keyneosoft. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController <UITextFieldDelegate,
    UITableViewDataSource, UITableViewDelegate> {
    IBOutlet UITableView *cityTableView;
    IBOutlet UITextField *cityNameTextField;
}

@property (strong, nonatomic) NSArray *cityList;

@end
```

```objc
#pragma mark - UiTextField delegate methods

- (void)textFieldDidBeginEditing:(UITextField *)textField {
    // on charge toute la base de données dans nos cellules
    CityDAO *cityDAO = [[CityDAO alloc] init];
    _cityList = [cityDAO getCityListFrom:@""];

    [cityTableView reloadData];

}

- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];
    return YES;

}

- (BOOL)textField:(UITextField *)textField
shouldChangeCharactersInRange:(NSRange)range
replacementString:(NSString *)string
{
    // on auto-complete le texte entré par l'utilisateur
    // on affiche les villes correspondant au texte entré

    NSMutableString *tmp = [[NSMutableString alloc]
                            initWithString:textField.text];
    [tmp replaceCharactersInRange:range
                       withString:string];

    CityDAO *cityDAO = [[CityDAO alloc] init];
    _cityList = [cityDAO getCityListFrom:tmp];

    [cityTableView reloadData];

    return YES;
}
```

FIGURE 2.12 – Delegate permettant la récupération des interactions utilisateurs sur un champ texte en Objective-C pour iOS

2.4.4 Définition du modèle

Dans notre application, le modèle consiste en deux classes présentées sur la figure 2.13. La classe **City** représente une ville dans notre application (nom de la ville et son code postal). Ensuite, nous avons une classe permettant d'accéder à notre base de données **CityDAO**. Étrangement, que l'on soit sur Android, iOS ou Windows Phone, les parties visibles du modèle sont les mêmes sur toutes nos plates-formes cibles à quelques exceptions près (types différents, paramètres d'entrée différents de certaines méthodes ...). Le traitement interne lui est complètement différent sur chacune des plates-formes cibles. Par exemple, sur Android,

nous utilisons l'API [12] ***android.database.sqlite*** pour accéder à notre base de données. Cette API est considérée comme une API de haut niveau. Par exemple, pour effectuer une requête SQL sur notre base de données, nous pouvons utiliser des méthodes qui nous permettent de se passer du SQL. Sur iOS, ce n'est pas le cas. Nous utilisons la librairie ***libsqlite3.dylib*** qui est considérée comme une librairie de bas niveau. En effet, nous devons effectuer des traitements beaucoup plus fins avec cette librairie. Par exemple, lorsque nous lançons une requête SQL, nous devons écrire nous même la requête SQL. Il faut ensuite lancer la requête et gérer les erreurs de syntaxe .... Sur Android, nous n'avons pas besoin de traiter ces erreurs, l'API génère elle-même les requêtes SQL. Sur Windows Phone 8, nous accédons à notre base de données avec une API de haut niveau comparable à Android.

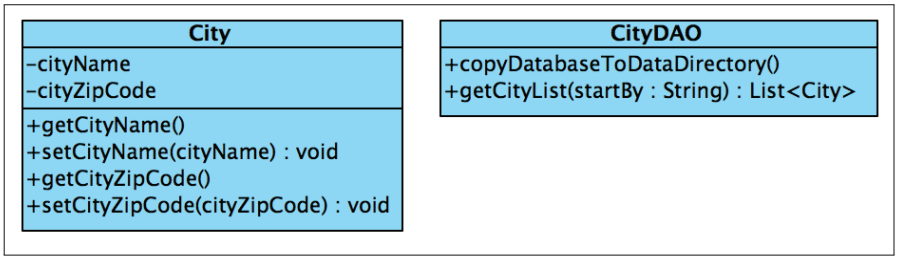


FIGURE 2.13 – Diagramme de classes : couche modèle de notre application d'autocomplétion

Au vue des différences relevées, il est clairement intéressant de cacher les contraintes techniques liées à chaque SDK cible lors du développement d'une application. Aujourd'hui, les développeurs ont besoin d'un temps d'adaptation à chaque changement de SDK. Ce temps d'adaptation comprend le changement d'IDE, de langage de programmation mais aussi les changements de façon de penser, de développer, d'utilisation d'APIs.

2.5 Attentes et Éxigences

Le développement d'applications mobiles est sujet à beaucoup de contraintes techniques liées à l'hétérogénéité des terminaux mobiles que ce soit au niveau des architectures matérielles ou que des architectures logicielles. Ces contraintes coûtent très cher aux entreprises. Il est donc devenu aujourd'hui nécessaire de baisser le coût de développement de ce type d'applications.

Dans les sections précédentes, nous avons identifié plusieurs difficultés liées au développement mobile multiplateforme. Certaines ne peuvent pas être atténuées comme, par exemple, les contraintes liées au déploiement d'une application mobile sur chacun des magasins d'applications officiels de chaque plate-forme cible. Nous avons identifié une phase sur laquelle il est possible d'influer pour baisser les coûts de développement d'une application mobile : la phase de création. Avec une application simple, nous avons remarqué qu'il y avait beaucoup de différences que ce soit sur la structure ou sur l'implémentation de certaines parties de l'application. Si nous arrivons à faciliter le passage d'une technologie à une autre, nous baisserons automatiquement le coût de développement. Les développeurs n'auront plus besoin d'avoir un long temps d'adaptation à chaque changement de plate-forme.

Les attentes d'une solution facilitant le développement d'applications mobiles multiplateformes sont multiples [CJR12]. Elles peuvent être décrites de deux points de vue complémentaires : les utilisateurs finaux des applications générées par ce type d'outil et les développeurs d'applications qui utilisent ce type d'outil.

12. API : Application Programming Interface

Nous voulons diminuer le temps de développement d'une application mobile en cachant toutes les parties spécifiques de chaque plate-forme. Les développeurs s'attendent donc à avoir un cadre commun pour concevoir, implémenter et/ou exécuter leurs applications. Cet outil devra leur permettre de gagner du temps. Ils devront écrire moins de lignes de code, peut être avoir un seul code source commun et non trois codes sources différents. Ensuite, cet outil devra permettre d'utiliser facilement les outils de développement existants. Nous voulons garder la puissance des environnements de développement actuels. De plus, les entreprises mobiles ne sont pas prêtes à abandonner tous les logiciels qu'elles ont réalisés depuis quelques années. Autant que possible, la solution trouvée devra offrir la possibilité de réutiliser le code source existant des entreprises. L'objectif est de ne pas re-développer des applications déjà implémentées. Ce cadre commun devra aussi respecter les normes fixées par chacun des magasins d'applications cibles et s'adapter à leurs évolutions. Les applications générées avec ce type d'outil doivent être déployables sur les magasins officiels. Après leur déploiement, les applications doivent être simples à maintenir. Enfin, l'outil devra s'adapter rapidement aux changements du marché. Si une nouvelle version d'un système d'exploitation cible est publiée ou si une nouvelle plate-forme apparaît, il faut que l'outil puisse prendre en compte ces modifications très rapidement. Pour autant, ce cadre commun ne doit pas léser les utilisateurs de smartphones. Dans la suite, nous listons toutes les exigences que ce type de solution doit respecter dans le but de ne pas réduire le périmètre des applications mobiles actuelles.

Les utilisateurs finaux ne veulent pas perdre le niveau d'expérience utilisateur fourni actuellement par le développement natif [13]. Dans le cas d'une application mobile, l'expérience utilisateur ne se limite pas à sa fluidité mais aussi aux interactions possibles avec l'environnement. Les smartphones sont principalement utilisés dans des situations de mobilité. Il faut prendre en compte toutes les contraintes engendrées par l'environnement (perte de réseau, peu de batterie ...). Pour faire face à cela, les smartphones intègrent une multitude de capteurs. Certains capteurs permettent, par exemple, de récupérer la localisation d'un utilisateur dans la rue via le GPS, mais aussi dans un bâtiment via des capteurs Bluetooth. Ainsi, une application est capable d'adapter son exécution aux changements de contexte. Dans le même ordre d'idée, il est possible de faire communiquer des smartphones les uns avec les autres soit de façon classique avec le bluetooth, soit de façon moins habituelle en mettant deux smartphones dos à dos (Android beam [14]). Enfin, les applications sont maintenant connectées avec l'extérieur. Elle peuvent recevoir des notifications ou enregistrer leurs données sur le cloud. Aujourd'hui, il est impossible de se passer de ces nouveaux usages, et par la même occasion de diminuer le niveau de l'expérience utilisateur des applications mobiles.

En plus des nouveaux usages fournis par les applications mobiles, leurs performances et leurs réactivités sont des parties conséquentes de la qualité de l'expérience utilisateur. Par exemple, si nous décidons que la solution pour simplifier le développement multiplateforme est d'intégrer un moteur d'exécution dans chaque application (celui-ci étant capable d'exécuter la même application sur n'importe quelle plate-forme), il faudra quand même que le lancement des applications soit instantané (temps de lancement du moteur d'exécution suivi du temps de lancement de l'application). Si ce n'est pas le cas, l'utilisateur aura alors une mauvaise évaluation de l'application et risque de la dés-installer très rapidement. Toujours en gardant cet exemple, les applications générées ne doivent pas être lourdes. Les smartphones sont souvent limités par la capacité du disque dur, habituellement entre 2 Go et 16 Go. Il ne faudrait pas que le moteur d'exécution pèse 1 Go alors que souvent les applications natives ne dépassent pas les 10 Mo. Ensuite, il est très important que l'application soit réactive. Par exemple, lorsqu'un utilisateur accède à un site web grâce au navigateur de son smartphone,

13. Développement natif : signifie que le développement d'une application a été effectué avec les outils officiels de la plate-forme visée

14. Vidéo présentant le principe d'Android beam : https ://www.youtube.com/watch ?v=4D8UVZzwADo

il n'est pas rare que lorsqu'il clique sur un bouton, il se passe entre cinq et dix secondes avant que la page web ne change d'état. Dans ce cas, il y a clairement un manque de réactivité du navigateur ou plus exactement de la page web. Ce problème ne doit pas se produire avec un outil permettant le développement d'applications mobiles. Les applications mobiles sont beaucoup plus utilisées que les navigateurs web grâce à leur réactivité.

Il est également indispensable de différencier les applications générées par ce type d'outil en fonction de chacune des plate-formes cibles. Prenons un exemple concret. Sur iOS, habituellement, la navigation entre les vues est implémentée à travers une pile de contrôleurs. Si l'utilisateur ouvre une page, son contrôleur s'ajoute à la pile. Si il clique sur le bouton retour de la barre de navigation, le contrôleur est enlevé de la pile et la vue précédente est rechargée, figure 2.14. Sur Windows Phone 8, ce système n'existe pas. Sur cette plate-forme, un des usages en terme de navigation est d'utiliser un panorama ou des onglets 2.15. L'utilisateur navigue alors grâce à des actions de swipe sur l'écran. Le risque, en utilisant un outil facilitant le développement d'applications multiplateformes, est de perdre ces différences de comportement. Dans ce cas, les utilisateurs n'auraient plus de raisons de choisir un smartphone avec tel ou tel système d'exploitation. Les applications auraient de toute façon le même comportement. Pour nous, ces différences entre systèmes d'exploitation doivent être sauvegardées, c'est ce qui permet de faire évoluer les usages. Aujourd'hui, chaque système d'exploitation fournit une gestion de l'expérience utilisateur différente [CLY14].

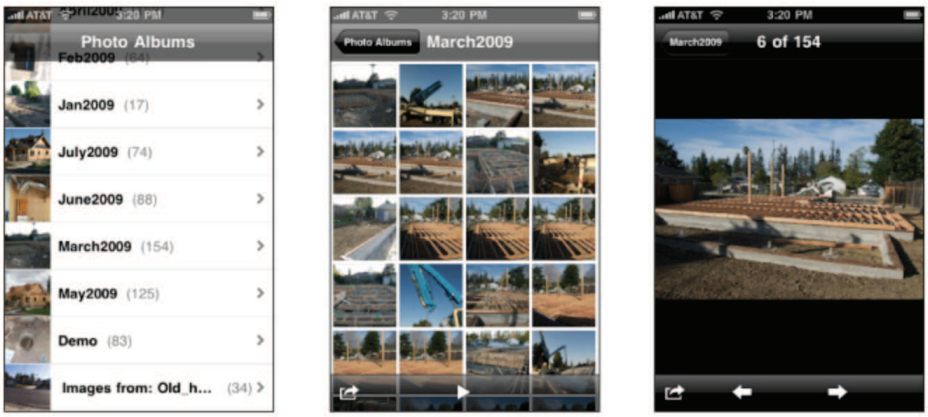


FIGURE 2.14 – Navigation par pile de vues sur iOS

Ensuite, si une fonctionnalité est disponible à travers l'un des systèmes d'exploitations cibles alors qu'elle ne l'est pas sur les autres, il faut permettre son utilisation. Par exemple, Android permet l'utilisation du NFC sur les smartphones équipés de cette technologie alors qu'iOS ne l'autorisait pas avant septembre 2014. Avec ce type d'outil, il faudra permettre l'utilisation du NFC et ne pas bloquer cette fonctionnalité même si elle n'était pas commune à toutes les plates-formes cibles.

Enfin, le marché du mobile peut changer très vite. Comme nous l'avons présenté au début de ce chapitre, pendant les 3 ans de cette thèse, deux plates-formes mobiles ont disparu (Symbian et BlackBerry OS) et une est apparue (Windows Phone). Une solution, telle que celle que nous aimerions avoir, doit impérativement pouvoir prendre en compte rapidement les changements du marché. Nous pouvons imaginer qu'iOS soit remplacé dans quelques années par un autre système d'exploitation. Il faudra pouvoir ajouter ce nouveau système à cette solution. De la même façon, une plate-forme déjà existante peut changer complètement son écosystème. Nous avons pu observer cela avec Apple qui a sorti un nouveau langage de programmation pour iOS : le swift. Cette nouveauté était inattendue d'autant qu'Apple est déjà à la huitième version d'iOS. Il faudra donc que la solution prenne rapidement en compte

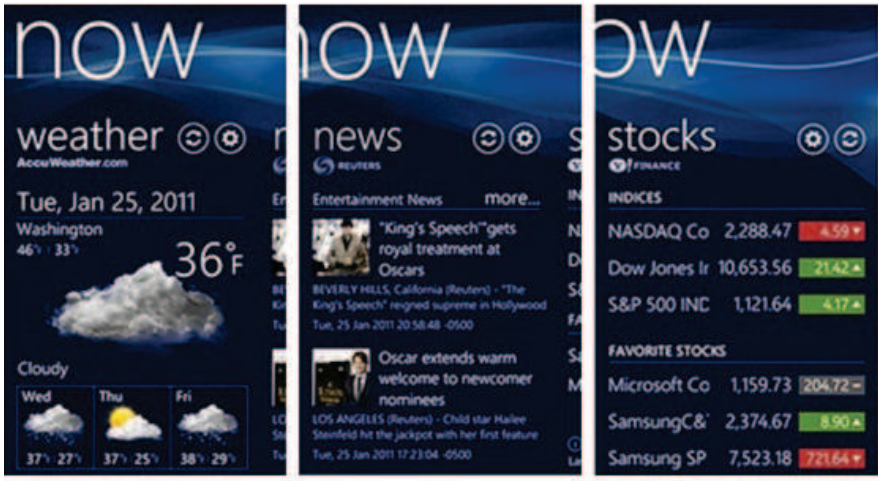


FIGURE 2.15 – Navigation par panorama sur Windows Phone 8

ce nouveau langage. Sans ça, la solution ne pourra pas être viable et durer dans le temps.

2.6 Conclusion

Le développement d'applications pour plusieurs plates-formes mobiles est très contraignant. Ceci est dû à l'hétérogénéité engendrée par les fournisseurs de systèmes d'exploitations. Ces différences se situent à trois niveaux distincts :

- Le déploiement des applications.
- Les outils de développement nécessaires à l'implémentation.
- Les techniques liées au développement des applications sur chacune des plates-formes visées.

Dans ce chapitre, nous avons montré qu'il était intéressant de cacher ou d'atténuer les différences techniques liées au développement des applications mobiles sur chacune des plates-formes cibles. Pour ce faire, nous avons implémenté une application simple sur iOS, Android et Windows Phone 8. Nous avons identifié de nombreuses différences que ce soit au niveau des outils utilisés (plusieurs environnements de développement, plusieurs langages de programmation, ...) ou que ce soit au niveau de l'utilisation de chacun des SDKs (APIs différentes, patrons de conception différents ...). Aujourd'hui, toutes ces différences font perdre un temps considérable aux développeurs. Pour baisser le coût de développement des applications mobiles multiplateformes, nous devons donc les atténuer.

Suite à cela, nous avons listé les exigences que nous attendons d'un outil permettant de baisser les coûts de développement d'une application mobile multiplateformes. Nous avons remarqué qu'il ne fallait pas brider les applications mobiles. Nous voulons fournir la même expérience utilisateur que ce que nous trouvons actuellement. Nous souhaitons également garder l'identité de chaque plate-forme cible ainsi que toutes leurs fonctionnalités respectives.

Les développeurs ont donc besoin d'un cadre commun de conception et/ou d'implémentation et/ou d'exécution. Cependant, ce cadre commun ne doit pas limiter l'utilisation des applications mobiles. Il doit aussi être facilement adaptable pour prendre en compte rapidement les changements liés au marché du mobile (montée de versions des systèmes d'exploitation

ou arrivée d'un nouveau système ou langage de programmation).

Dans le prochain chapitre, nous étudions les différents outils de développement existants, permettant de baisser les coûts de développement d'applications mobiles multiplateformes.

Chapitre 3

Environnement de développement pour applications mobiles multiplateformes

Dans ce chapitre, nous présentons plus précisément Android, iOS et Windows Phone 8. Nous présentons notamment leur architecture interne pour se focaliser sur les zones d'exécution de chacun de ces systèmes d'exploitation. Ensuite, dans la deuxième partie, nous nous focalisons sur des solutions permettant le développement d'applications multiplateformes. Les solutions étudiées se basent sur les systèmes d'exploitation mobiles, soit pour unifier le développement, soit l'exécuter. Nous considérons donc que ces solutions sont proches des systèmes d'exploitation existants. Dans le chapitre suivant, nous étudierons les solutions qui font complètement abstractions des systèmes d'exploitation existants.

Sommaire

3.1 Introduction 25
3.2 Plusieurs environnements mobiles 26
3.2.1 Android 26
3.2.2 iOS 28
3.2.3 Windows Phone 8 29
3.3 Solutions pour le développement multiplateforme proches des systèmes d'exploitations mobiles 31
3.3.1 Compilateur source à source 31
3.3.2 Applications web 35
3.4 Conclusion 41

3.1 Introduction

Dans l'idéal, les développeurs mobiles aimeraient une solution qui leur permette d'implémenter une seule fois leurs applications mobiles dans un seul langage. Avec ce genre de

solution, ils ne seraient plus obliger de se former à plusieurs technologies, d'écrire plusieurs fois le même code source avec des langages et SDK différents, de maintenir plusieurs codes sources différents. Le gain économique serait alors considérable pour les entreprises de développement mobile. Cependant, ces solutions ne doivent pas limiter l'expérience utilisateur. C'est aujourd'hui l'expérience utilisateur qui permet à une application d'être un succès ou non. Pour ne pas la brider, il faut que l'interface utilisateur soit fluide et que les performances de l'application soient correctes. De plus, les applications mobiles doivent sans cesse se renouveler en suivant les tendances du marché et les nouvelles technologies fournies par les systèmes d'exploitation (utilisation du NFC, des iBeacons, etc.).

Plusieurs travaux venant du monde de la recherche ou industriel permettent d'implémenter une application mobile une seule fois pour la déployer sur plusieurs plates-formes cibles [PSC12, HHM12a, HHM12b, DDBN13, RdS12]. Dans ce chapitre, nous passons en revue certaines de ces solutions. Nous nous focalisons plus particulièrement sur les solutions pouvant être considérées comme proche des systèmes mobiles existants. Les autres solutions seront, quant à elles, étudiées dans le prochain chapitre. Avant de commencer cette étude, section 3.3, nous revenons sur l'architecture des systèmes d'exploitation mobiles les plus populaires : Android, iOS et Windows Phone, section 3.2.

3.2 Plusieurs environnements mobiles

Avant d'étudier les solutions permettant de développer une application mobile multiplateforme, il est intéressant d'étudier plus en détail, l'architecture des principaux systèmes d'exploitation mobiles. Dans le chapitre précédent, nous avons déjà listé plusieurs différences entre Android, iOS et Windows Phone 8. Ici, nous nous focalisons sur l'architecture interne de chaque solution. C'est à partir de cette architecture et de l'approche choisie par chacun des fournisseurs de ces systèmes d'exploitation que nous pourrons comprendre les impacts de chacune des solutions étudiées.

3.2.1 Android

Android [HKM10] a été spécifié par un consortium : Open Handset Alliance [1] (OHA) créé en novembre 2007 à l'initiative de Google. Aujourd'hui, l'OHA compte 80 membres dont des opérateurs de téléphonie mobile, des fabricants de matériel, des fabricants d'appareils mobiles, des concepteurs de logiciels ainsi que des acteurs de la commercialisation. Leur objectif était de créer un système complet, ouvert et gratuit pour les mobiles. Aujourd'hui, c'est le leader en matière de système d'exploitation mobile (70% du marché européen).

Architecture

Android n'a pas été conçu uniquement pour les smartphones mais pour tous les appareils embarqués tels que des tablettes, des montres, des baladeurs, des télévisions, des voitures, etc. en plus des smartphones. Pour ce faire Android est basé sur un Kernel Linux qui peut être installé sur n'importe quel appareil. Le système d'exploitation d'Android est composé de 5 couches, figure 3.1 [2] :

— La couche la plus basse du système est basée sur un **noyau Linux**. Ce noyau gère des services systèmes tels que la sécurité, la gestion de la mémoire, la gestion des

1. Site officiel de l'Open Handset Alliance : http ://www.openhandsetalliance.com
2. Image récupérée du site d'Android : http ://source.android.com/devices/

processus .... C'est aussi elle qui fait le lien entre le matériel et la partie logicielle (driver pour la caméra ...). Pour cela, les drivers sont connectés à la couche supérieure. Les drivers sont implémentés par les constructeurs qui souhaitent installer Android sur leur appareil. Par exemple, Samsung a ses propres drivers qui feront le lien entre Android et son matériel.

— La couche suivante correspond à une couche d'**abstraction matérielle**. Cette couche fait le lien entre les services Android et le noyau Linux. C'est sur ces interfaces que les constructeurs voulant intégrer Android dans leurs appareils doivent connecter leurs drivers matériels.

— La troisième couche correspond aux **services** fournis par Android. C'est en quelque sorte les APIs qui sont accessibles à partir de nos applications mobiles. Elles sont divisées en deux groupes : les APIs liées aux médias (lecture, enregistrement ...) et les APIs liées au système (gestion des vues affichées, des applications etc.).

— La couche **Binder IPC** permet de faire le lien entre les applications (la dernière couche du système) et les services Android.

— La couche **application et frameworks** contient des applications pré-installées telles qu'un navigateur web, un client email, un service de SMS, un calendrier ainsi que toutes les applications pouvant être installées à travers play store, le market officiel d'applications d'Android.

Sur la figure 3.1, nous ne voyons pas comment les applications sont exécutées. En fait sur Android, c'est une machine virtuelle qui s'occupe de cela : **Dalvik VM**. Cette machine virtuelle a été développée pour les systèmes embarqués avec comme objectif d'être rapide. Pour ce faire, ils ont introduit un nouveau format de bytecode appelé Dalvik.

Ouverture

En plus d'être un logiciel open-source, Android est un système d'exploitation libre [3]. Il est distribué sous la licence Apache 2.0 [4]. La grande force d'Android est donc son ouverture. Par exemple, n'importe quel constructeur ou opérateur de téléphonie mobile peut modifier Android puis l'installer sur un de ses smartphones. Chacun pourra alors, adopter son propre style, installer ses propres applications etc. Cela explique son adoption par un grand nombre de constructeurs et opérateurs mobiles. De plus Android est gratuit.

Cette ouverture est une force et une faiblesse en même temps. En effet, les périphériques sur lesquelles Android sont installés n'ont pas les mêmes composants matériels (écrans différents, processeurs différents, etc.). Pour les utilisateurs finaux c'est une force. Ils ont le choix entre plusieurs smartphones très différents. Néanmoins c'est une faiblesse pour les développeurs. Cela les oblige à tester leurs applications sur plusieurs smartphones différents (achat massif de smartphones) ainsi que de gérer pendant la phase de développement toutes les différences qu'il pourrait y avoir. Par exemple, la gestion des différentes tailles d'écrans fait perdre beaucoup de temps aux développeurs alors qu'ils aimeraient se focaliser sur le développement d'innovations. Pour finir, dans le chapitre précédent, section 2.2, nous avons aussi montré que l'hétérogénéité des versions d'Android déployées sur le marché ne permettent pas aux développeurs d'implémenter leurs applications pour la dernière version du système.

3. Un logiciel libre offre 4 libertés à l'utilisateur : 1) liberté d'exécuter le programme pour tous les usages, 2) liberté d'étudier le fonctionnement du programme, 3) liberté de modifier le programme, 4) liberté de redistribuer des copies du programme ou de ses modifications. Un logiciel open-source ne réunit souvent que 2 ou 3 de ces libertés.

4. licence Apache : autorise la modification et la distribution du code sous toutes formes (libre ou propriétaire, gratuit ou commercial)

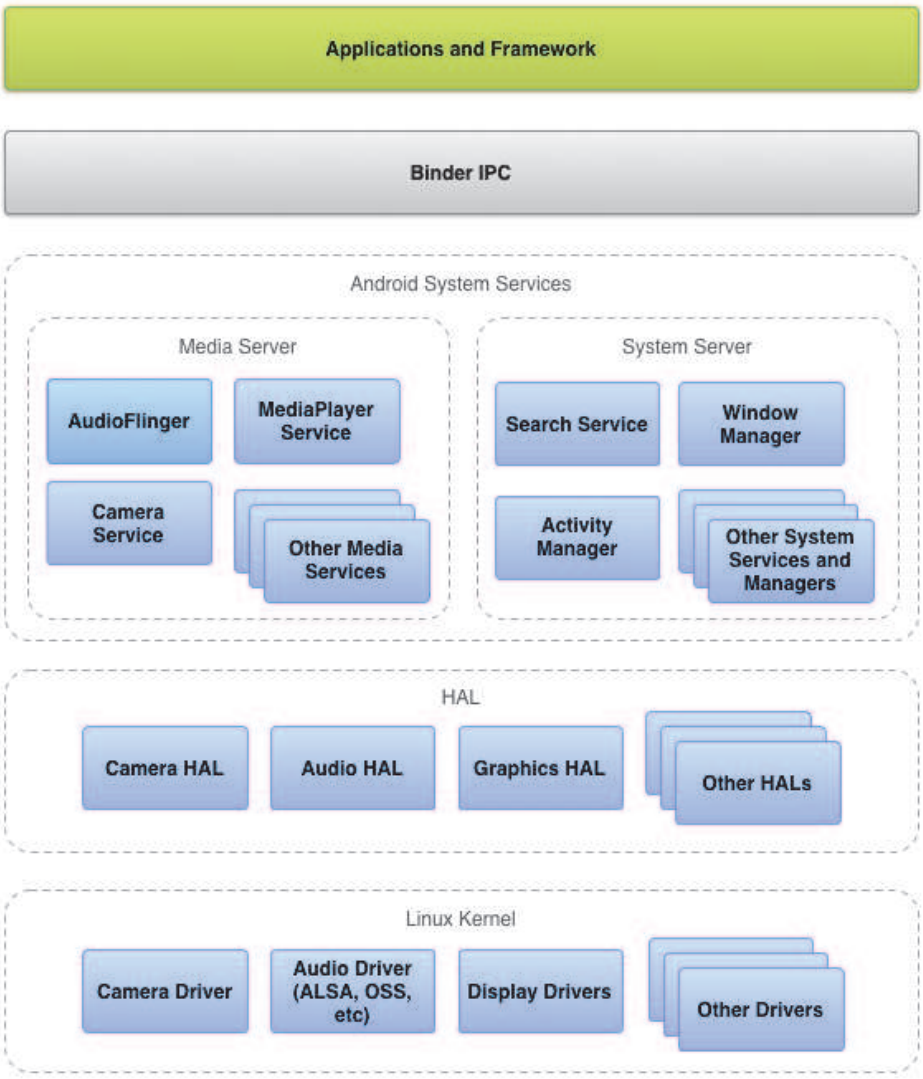


FIGURE 3.1 – Architecture du système d'exploitation Android

3.2.2 iOS

iOS [Neu13] est le système d'exploitation d'Apple installé sur iPhone, iPod Touch, iPad et Apple Watch. Apple a été le premier à lancer un smartphone grand public avec le succès qu'on lui connaît : l'iPhone. Ce smartphone intégrait un écran multi-touch, un accéléromètre et la localisation par triangulation GSM.

Architecture

iOS est une extension de Mac OS X. Son architecture est composée de quatre couches d'abstraction logicielle, figure 3.2 [5] :

— **Core OS** et **Core services** contiennent toutes les interfaces fondamentales d'iOS ainsi que les interfaces qui permettent la gestion des fichiers, la gestion des sockets... Ces interfaces sont accessibles par différents frameworks tels que "Accelerate Framework" qui gère les mathématiques, ou encore "Core Data Framework" qui gère les données...

5. Image récupérée du site d'Apple : https ://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneO CH1-SW1

Ces différents frameworks ont été optimisés pour le matériel sur lequel iOS fonctionne.
— la couche **Media** contient les technologies fondamentales pour gérer les graphiques 2D et 3D ainsi que les vidéos et l'audio.
— La couche **Cocoa Touch** (écrite en Objective-C) fournit tous les frameworks utiles à la création d'une application iOS (implémentation des vues ...).

iOS n'a pas besoin d'une machine virtuelle pour exécuter une application car il est installé sur des périphériques homogènes. De plus, Apple se réserve le droit d'installer iOS uniquement sur ses appareils. Il est donc inutile pour eux de montrer comment faire le lien entre le système d'exploitation et le matériel. Bien que le système utilise toujours les mêmes bases, iOS évolue rapidement (une nouvelle version par an). Chaque version faisant évoluer les APIs de développement, les applications ont besoin d'être mises à jours assez souvent. Avec iOS version 8, Apple a par exemple introduit un nouveau langage : le Swift.

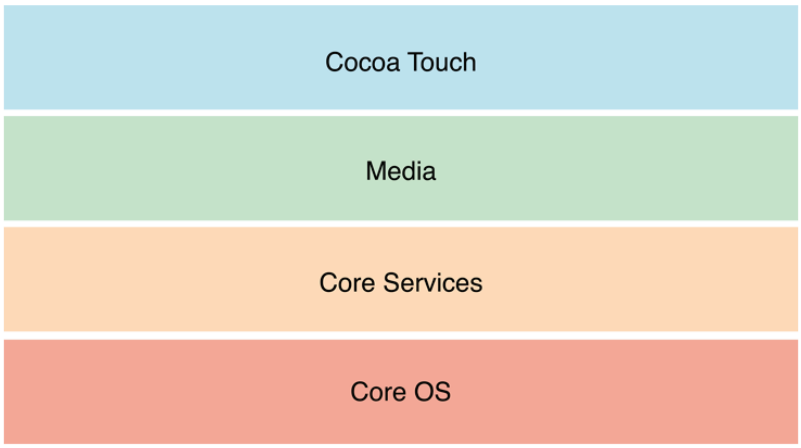


FIGURE 3.2 – Architecture du système d'exploitation iOS

Ouverture

Contrairement à Android, iOS est propriétaire. En effet, il n'est pas possible de modifier son code source. L'intégralité du code source d'iOS n'est d'ailleurs pas disponible. Ensuite, les appareils tournant sous iOS sont exclusivement fournis par Apple ce qui oblige les opérateurs à se plier à cette technologie. Aujourd'hui, il n'y a eu que 10 modèles d'iPhone différents. Malgré ceci, les appareils d'Apple sont vendus massivement et appréciés par ses utilisateurs. Par conséquent, les problèmes de fragmentation d'Android ne sont pas applicables à iOS, les appareils d'Apple étant homogènes au niveau des composants matériels, le déploiement d'applications est plus simple. Les développeurs iOS n'ont pas besoin de s'inquiéter de savoir si leur application va fonctionner sur tel ou tel type d'appareils. Dans 95% des cas, une application iOS fonctionne sur tous les smartphones Apple sans aucune modification.

3.2.3 Windows Phone 8

Windows Phone 8 [LT13] a été conçu par Microsoft. Il a été lancé le 29 octobre 2012, 2 ans environ après la première version pour grand public : Windows Phone 7. Windows Phone 7 fait suite à Windows Mobile qui, lui, était conçu pour les entreprises.

Architecture

L'architecture de windows phone 8, figure 3.3 [6], est très proche de l'architecture de Windows 8. En effet, windows 8 permet d'exécuter deux types d'applications : des applications de bureau et des applications provenant du magasin d'applications windows, le windows store. Elle est divisée en 3 couches distinctes :

- La couche Kernel fait le lien entre le matériel et la partie logicielle. C'est dans cette couche que les drivers des différents capteurs seront connectés au système.
- La couche intermédiaire contient deux sous-couches : les librairies de WinRT et les moteurs d'exécution d'applications. Au niveau des librairies, les développeurs peuvent trouver toutes les bibliothèques utiles au développement d'applications (Interface utilisateur, stockage de données, etc.).
- Enfin, la couche application contient toutes les applications pouvant être récupérées dans le magasin d'applications. Elles sont exécutées par la machine virtuelle CLR (Common Language Runtime) disponible dans la couche inférieure du système. Cette machine virtuelle permet d'exécuter des applications .Net, C#, etc.

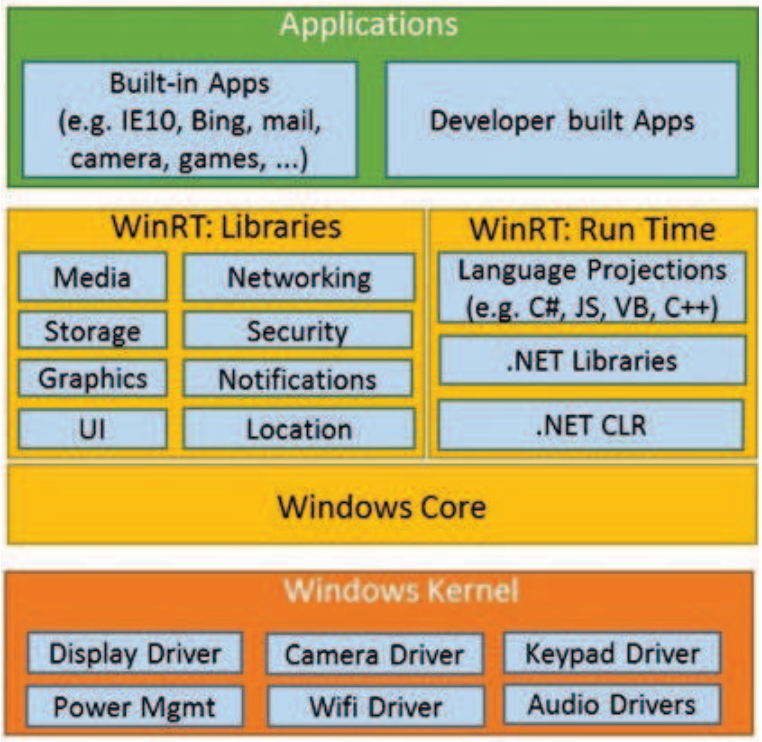


FIGURE 3.3 – Architecture du système d'exploitation Windows Phone 8

Ouverture

Comme iOS, Windows Phone 8 est propriétaire. Cependant, la politique économique de Microsoft est différente. En effet, les constructeurs de smartphones peuvent installer Windows Phone sur leurs appareils. Ainsi, les smartphones Nokia et HTC et même Samsung ont des smartphones avec Windows Phone comme système d'exploitation. Pour tirer parti au mieux du système d'exploitation, Microsoft oblige les constructeurs de smartphones à respecter

6. Image récupérée du site de Windows : http ://blogs.msdn.com/b/hanuk/archive/2013/04/18/introducing-windows-8-for-android-developers-part-1.aspx

certaines fonctionnalités [7]. Par exemple, les smartphones Windows Phone doivent avoir une caméra arrière alors que la caméra frontale est optionnelle. Le NFC est optionnel. Avec ces règles, les développeurs d'applications Windows Phone savent ce qu'il est possible de faire pour les applications grand public. Cela facilite le développement.

Pour récapituler, comme nous l'avons montré dans le chapitre précédent et, ensuite, avec l'architecture des systèmes d'exploitation mobiles Android, iOS et Windows Phone 8, les trois technologies sont très différentes. D'un côté, Android et Windows Phone 8 sont conçus pour fonctionner sur une multitude d'appareils différents alors qu'iOS ne fonctionne que dans les appareils Apple. Ce choix économique se ressent sur l'architecture de ces systèmes. Les applications Android et Windows Phone 8 sont exécutées à travers des machines virtuelles alors que sur iOS, les applications sont directement exécutées par le système sans intermédiaire. L'objectif d'Apple est d'avoir les smartphones et le système d'exploitation le plus performant possible.

3.3 Solutions pour le développement multiplateforme proches des systèmes d'exploitations mobiles

Dans cette section, nous nous focalisons sur les solutions permettant de développer une fois une application mobile pour ensuite, soit la générer pour plusieurs plates-formes cibles, soit pour l'exécuter à travers un navigateur web déjà présent sur chacune des plates-formes cibles. Nous considérons les solutions présentées comme des solutions proches des systèmes mobiles existants.

3.3.1 Compilateur source à source

Une première solution est basée sur des compilateurs sources à sources. Ce type de compilateur permet de transformer un code source écrit dans un langage de programmation de haut niveau en un autre code source écrit lui aussi dans un langage de programmation de haut niveau. Nous pouvons utiliser ce type de compilateur pour répondre à notre problématique, figure 3.4. Par exemple, nous pouvons imaginer, avec ce type de solution, écrire une application en C++ et ensuite qu'elle soit transformée en Java pour Android, en Objective-C ou Swift pour iOS et C# pour Windows Phone 8. Ensuite, le code source généré est compilé avec les outils fournis par chaque plate-forme cible dans le but de construire l'application finale pour chacun des systèmes d'exploitation cibles. L'objectif principal de la solution, est d'unifier le langage de programmation de l'application et ensuite de générer l'application finale pour toutes les plates-formes cibles. Ainsi, les développeurs d'applications mobiles n'auraient plus besoin d'implémenter plusieurs versions de la même application avec des technologies différentes. Dans ce cas, le bénéfice est considérable. Les développeurs n'ont plus besoin d'être formés à plusieurs technologies et gagnent au maximum autant de fois le temps qu'il aurait mis pour développer plusieurs fois l'application. Plus il y a de plates-formes visées, plus le gain de temps est considérable. De plus, la maintenance elle aussi devient beaucoup plus simple. Il n'y a plus qu'un seul code source à maintenir.

Plusieurs solutions ont été mises en place dans ce sens. Parmi ces solutions, nous pouvons les séparer en deux groupes :

7. Spécifications du minimum requis pour installer Windows Phone sur un appareil : https ://dev.windowsphone.com/en-US/OEM/docs/Phone_Bring-Up/1.2__Windows_Phone_Chassis_requirements_summary

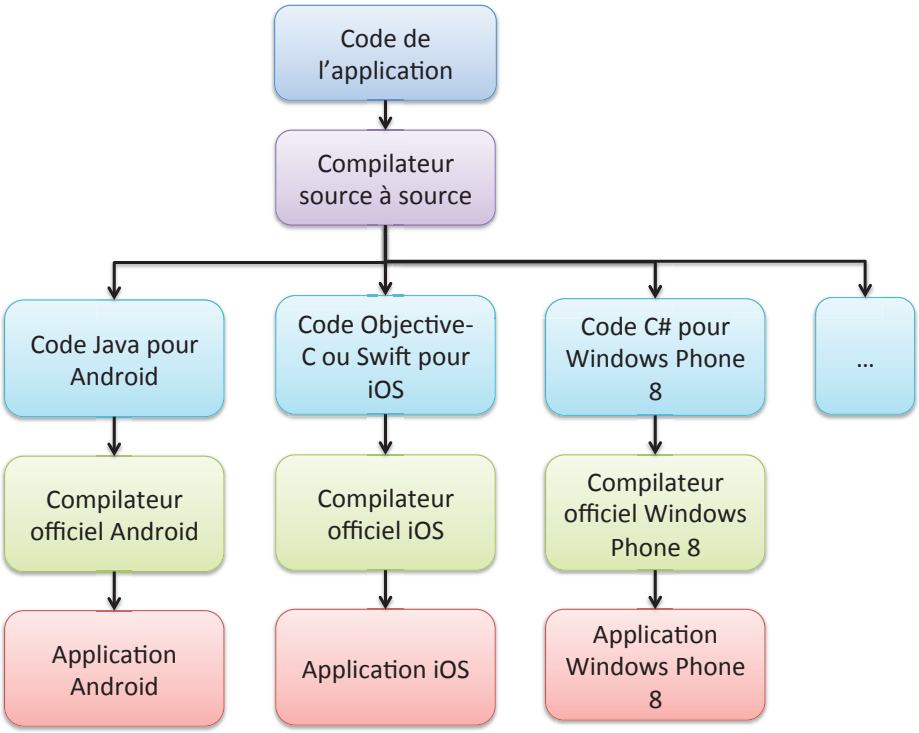


FIGURE 3.4 – Principe de la compilation source à source application au développement mobile multiplateforme

— Celles qui se basent sur un environnement de développement mobile existant pour ensuite générer plusieurs applications pour d'autres environnements mobiles.
— Celles qui se basent sur un nouvel environnement de développement pour implémenter une application mobile qui sera ensuite déclinée pour plusieurs plates-formes mobiles. Dans ce genre de solution, le challenge est de définir un environnement de développement commun qui réunit toutes les fonctionnalités de toutes les plates-formes mobiles existantes. Il ne faut pas limiter les développeurs d'applications.

Transformer une application implémentée pour une plate-forme mobile spécifique sur d'autres plates-formes mobiles

Une première solution est d'utiliser un environnement déjà existant et de transformer les applications implémentées pour cet environnement pour d'autres environnements. **J2ObjC**, **Java2CSharp**, **XMLVM** et **Xamarin** exploitent cette approche.

j2ObjC[8] et **Java2CSharp**[9] permettent de transformer une partie d'un code source Java respectivement en Objective-C ou en C#. Ce genre d'outils peut clairement être utilisé pour le développement d'applications mobiles multiplateformes. Nous pouvons imaginer créer une partie du code source en Java pour Android et ensuite le porter sur iOS et Windows Phone 8. Cependant, aujourd'hui, ces outils se focalisent uniquement sur la transformation du code source générique d'une application, c'est-à-dire, la représentation des données, qui est indépendante d'une plate-forme cible. Dans ces outils, seule la transformation interlangage a été implémentée. Les liens entre les APIs du langage de départ et des plates-formes cibles ne sont pas traités. L'outil ne peut donc pas transformer un appel à une API Android en

8. Blog J2ObjC : http ://j2objc.blogspot.fr/2014/01/android-and-iphones-and-web-oh-my.html
9. Proposition du projet Java2CSharp pour eclipse : http ://www.eclipse.org/proposals/c-sharp/

Objective-C ou Windows Phone. Contrairement à ces deux solutions, **XMLVM** gère les liens entre les APIs Android et iOS.

En effet, les concepteurs d'**XMLVM** permettent avec cette solution de porter une application complète Android sur iOS [Pud10] ou sur Windows Phone 7 [ANP11, PA13]. Pour ce faire, ils ne se basent pas directement sur le code Java de l'application Android mais sur le code compilé de celle-ci (le bytecode). Ils parsent le bytecode et le transforme en XMLVM (langage basé sur le XML [10]). Les fichiers XMLVM sont alors transformés dans le langage cible grâce à des fichiers XSLT. Le principal problème est la compatibilité entre les APIs Android et celles de la plate-forme cible. Ils ont donc redéfini les APIs Android pour qu'elles soient compatibles avec les APIs des autres plates-formes. Ces APIs ainsi redéfinies sont utilisées uniquement à la compilation de l'application et non lors du développement de la version Android.

Cette solution a, cependant, certains manques. En effet, elle prend en compte les APIs Android comme la base du développement de l'application. Cela signifie que les fonctionnalités spécifiques à iOS ou Windows Phone ne sont pas accessibles (iCloud, paiement in-app, etc.). De la même façon, les fonctionnalités spécifiques à Android ne peuvent pas être transformées sur iOS ou sur Windows Phone. Le développeur devra donc adapter le code généré pour iOS ou Windows Phone en fonction de ses besoins sur ces deux plates-formes. Il pourra ajouter les fonctionnalités manquantes. Par conséquent, les applications générées seront difficiles à maintenir. À chaque fois que le développeur ajoutera une évolution à son application dans son code Android, il devra re-générer son application et, par conséquent, refaire les liens avec les fonctionnalités spécifiques à iOS ou Windows Phone qu'il aurait déjà ajoutées. Cette solution permet donc de gagner du temps au départ de la phase de développement de l'application. Elle permet de développer une base commune à Android, iOS et Windows Phone. Cependant, par la suite, pour la maintenance, il sera très difficile d'utiliser encore cette solution.

Enfin, ce genre de solution est très difficile à mettre en place. Il faut pour chaque librairie "commune" à Android, iOS et Windows Phone développer une interface de compatibilité et ensuite fournir les règles de compilation pour chacune de ces librairies. Pour ne pas faire cette étape manuellement, les concepteurs d'XMLVM ont créé un outil permettant de générer à partir des interfaces des APIs Objective-C iOS des squelettes d'APIs implémentées en Java [PD12]. Même automatisé, c'est un travail considérable à effectuer surtout lorsque l'on sait que les APIs sont susceptibles de changer très rapidement. De plus, Apple vient de sortir Swift un nouveau langage de programmation pour iOS. Il faudra mettre à jour l'outil de génération de squelettes d'APIs pour prendre en compte ce nouveau langage. Cela peut prendre un certain temps.

Xamarin [11] [xam14] permet aux développeurs mobiles d'implémenter leurs applications en C#. Le code source est quand même divisé en fonction des plates-formes cibles. Par exemple, si le développeur veut implémenter une application pour Android et iOS, il aura deux codes sources différents. Bien sûr, une partie de ces deux codes sources sera commune. Cette partie contient tous les services qui sont indépendants d'une plate-forme cible tels que l'accès aux bases de données, la définition des modèles, l'envoi de requêtes HTTP, etc. Le reste de l'application, c'est-à-dire, l'interface utilisateur et l'accès aux capteurs matériels etc. est implémenté en utilisant un SDK propre à chacune des plates-formes cibles : **MonoAndroid** pour Android et **MonoTouch** pour iOS. Ces deux SDK représentent en fait le SDK des deux plates-formes cibles. Toutes les classes, interfaces, méthodes, etc. d'Android sont par exemple représentées dans MonoAndroid sous le langage c#. Ainsi, le développeur peut appeler toutes les méthodes du SDK Android en C#. Il peut exploiter les fonctionnalités natives du SDK.

10. XML : Extensible Markup Language, langage à base de balises
11. Site web officiel de Xamarin : http ://xamarin.com

Il peut, par conséquent fournir une expérience utilisateur Android qui s'intégrera parfaitement dans Android. MonoTouch fournit quant à lui la version iOS du SDK en C#. Pour la partie commune, Xamarin fournit des librairies de haut niveau dont l'implémentation se base sur les deux SDK MonoAndroid et MonoTouch. Lors de la compilation, le code source C# pour Android ou iOS est transformé en code natif. Vu que les codes sources appellent la représentation des méthodes natives, les règles de compilation sont plutôt simples. Il faut uniquement transformer le code C# en langage Java ou Objective-C. Nous présentons plus en détail Xamarin dans le chapitre 10, section 10.2.3.

Fournir un unique SDK pour le développement mobile qui réunit les SDKs de chaque plate-forme mobile existante.

Plusieurs autres solutions permettent de développer une application mobile multiplateforme à partir d'un seul code source commun. Contrairement aux solutions présentées précédemment, le langage de programmation et les APIs de programmation utilisées ne dépendent pas des environnements de développement natif (Android, iOS, etc.). L'objectif est de fournir une solution complètement indépendante de toutes les plates-formes cibles. Cependant, le but final est quand même de fournir toutes les fonctionnalités de toutes les plates-formes mobiles. Ce genre de solution réunit donc les fonctionnalités de chacun des SDKs natifs sous un seul SDK.

MoSync SDK [12] permet d'écrire une application en C++ qui est ensuite transformée en langage natif pour Moblin et les machines virtuelles Java ME. Pour ce faire, les concepteurs ont redéfini des librairies C++ pour la création de l'interface utilisateur, pour l'accès aux capteurs matériels, pour le stockage, etc. Cependant, l'interface utilisateur générée était la même sur toutes les plates-formes cibles. Il n'était pas possible d'utiliser l'aspect natif de la plate-forme cible. De plus, l'interface utilisateur ne ressemblait pas à ce qui est fait habituellement dans une application mobile (des listes, des champs textes, etc.). Enfin, les performances de cette solution étaient mauvaises [13]. Cette solution avait été initialement implémentée pour répondre aux besoins de standardisation de Java ME. Nous revenons sur le Java ME dans le chapitre 4, section 4.3.1. Aujourd'hui, avec les évolutions du mobile, MoSync a modifié son environnement de développement dans le but de développer des applications mobiles à partir des technologies web. Nous revenons sur cette approche dans la section 3.3.2. Ce qu'il faut retenir est que les concepteurs de MoSync ont préféré arrêter l'approche compilateur source à source, pour se consacrer à une approche basée sur les technologies web. Ceci est principalement dû à la difficulté de faire le lien entre les APIs du SDK commun avec les SDKs de chaque plate-forme cible.

Avec **NeoMAD** [14] les développeurs implémentent leurs applications en Java. Cependant, le code source n'a aucun lien avec le SDK Android. En effet, NeoMad est fourni avec plusieurs librairies qui s'occupent chacune d'un aspect des smartphones. Toutes les APIs de NeoMAD sont indépendantes de toutes les plates-formes cibles. Seule les librairies pour le développement de l'interface utilisateur sont inspirées d'Android. Après avoir développé l'application, le code source est compilé par le compilateur NeoMAD et est transformé pour plusieurs plates-formes cibles telles qu'Android, iOS, Windows Phone, etc.

Ce genre de solutions est difficile à mettre en oeuvre. En effet, il faut redéfinir toutes les APIs essentielles au mobile en ne se basant sur rien. Par exemple, il faut redéfinir les

12. MoSync : http ://www.mosync.com/docs/sdk/index.html
13. Bilan sur les librairies de création d'interface d'utilisateur de MoSync : http ://www.mosync.com/docs/sdk/cpp/guides/ui/creating-user-interfaces-mosync/index.html
14. Site web officiel de NeoMAD : http ://neomades.com/fr/

APIs de bas niveau telles que le rapatriement des informations sur le smartphone (batterie, carte SIM, adresse IP etc.), la récupération des valeurs des différents capteurs (accéléromètre, géolocalisation etc.) etc. Il faut aussi redéfinir les APIs de plus haut niveau telles qu'un scanner de code barre. Aujourd'hui, ce genre d'APIs est disponible chez Apple [15]. En fin de compte, il faut réécrire un SDK mobile que les fournisseurs de système d'exploitation ont développé depuis plusieurs années. Pour compliquer encore plus la tâche, il faut faire le lien entre le nouveau SDK et plusieurs SDK mobiles existants. Cette étape ne peut pas être automatisée. Le développement de ce genre de solutions peut donc être très long et très difficile à finaliser. Le risque est, bien sûr, qu'un nouveau système d'exploitation voit et le jour et que ce type de solution ne soit jamais capable d'être lié avec ce nouveau système.

Discussions

Pour récapituler, les solutions basées sur les compilateurs source à source ne répondent pas à tous nos besoins. En effet, les solutions qui permettent de générer une partie d'une application à partir d'un seul code source ne permettent pas de maintenir efficacement les applications générées. Nous devons pour chaque évolution qui touche à la partie commune de l'application la re-générer, l'intégrer dans le code source de l'application cible et refaire les liens entre la partie évoluée et la partie déjà présente dans l'application spécifique à la plate-forme cible. Quant aux solutions basées sur un langage commun et accompagnées d'un nouveau SDK, elles ne permettent pas de suivre facilement les évolutions du domaine du mobile. Seul Xamarin permet de créer une application Android et iOS en respectant nos besoins. Cependant, nous verrons dans le chapitre 10, que les performances fournies par les applications générées par Xamarin ne sont pas suffisantes pour nous.

3.3.2 Applications web

Depuis quelques années, les technologies du web pour le développement mobile se sont démocratisées [LGA10]. En effet, selon Vision Mobile, 52% des développeurs mobiles ont utilisé HTML 5 pour développer leurs logiciels [vis13]. Parmi ces 52%, 38% sont dédiés au développement de sites web mobiles. Ce genre de sites sont souvent développés en suivant les règles du responsive design [Fra12] qui permettent d'implémenter des sites optimisés pour toutes les tailles d'écrans possibles que ce soit pour des smartphones, des tablettes ou des ordinateurs de bureau. Le choix d'un site web est donc devenu un bon compromis pour avoir une visibilité sur plusieurs plates-formes mobiles. En effet, un site web est développé une seule fois à partir de langages de programmation web HTML 5, CSS, JavaScript, PHP etc. et fonctionne sur Android, iOS, Windows Phone etc. Cependant, les navigateurs web ne fournissent pas la même expérience utilisateur qu'une application native. Ils n'ont pas, par exemple, accès à tous les capteurs matériels du smartphone. Pour pallier à ce manque, le World Wide Web Consortium (W3C) travaille sur des spécifications communes d'accès aux APIs natives à travers les navigateurs web. Aujourd'hui, les spécifications matures pouvant être implémentées dans les navigateurs web mobiles concernent l'accès aux fonctionnalités audio [AWR13], l'accès aux médias (caméra et micro) [KOHM14], l'accès à l'accéléromètre [TVBP14], l'accès aux services de géolocalisation [Pop14], l'utilisation de bases de données locales [MSG+13], l'envoi de SMS, MMS [HMCF+14] etc. L'objectif est de fournir les mêmes fonctionnalités que les applications natives. C'est ensuite aux fournisseurs des navigateurs d'implémenter ces spécifications. Cependant, comme pour ce qui est des navigateurs web pour ordinateur de bureau, les fournisseurs de ces navigateurs n'implémentent pas toujours

15. API de scan d'Apple : https ://developer.apple.com/library/ios/documentation/AVFoundation/Reference/AVMetadata

toutes les spécifications du W3C ou lorsqu'ils les implémentent, l'accès aux APIs web peut changer en fonction du navigateur. Il faut alors que les développeurs de site web différencient leurs codes sources en fonction du navigateur sur lequel leurs sites web sont exécutés. La partie du code à modifier est cependant minime par rapport au développement total d'un site web.

Pour résoudre ce manque de compatibilité entre navigateurs, une nouvelle approche consiste à fournir un service installé sur le smartphone accessible depuis un site web à travers des requêtes HTTP [PTM14]. Ce service joue en fait le rôle d'une passerelle entre les sites web et les capacités matérielles et logicielles des smartphones. Le site web appelle le service avec une requête HTTP vers un port local. Cela a comme effet d'activer le service. Le service effectue alors l'action demandée (prendre une photo, récupérer le niveau de la batterie, récupérer les contacts, etc.). Ce nouveau processus adresse de nouvelles perspectives. Classiquement, lorsque nous installons une application native sur nos smartphones, la plate-forme prévient l'utilisateur des actions possibles de l'application, par exemple, la récupération des contacts. Dans cette nouvelle approche, un site web pourrait récupérer les contacts sans pour autant que l'utilisateur soit prévenu. Cela pose certains problèmes de sécurité liés aux données de l'utilisateur.

Dans le même ordre d'idée, **MoSync** qui avait auparavant publié un compilateur source à source, section 3.3.1, a lancé une solution basée sur les technologies web : **MoSync Reload** [16]. L'objectif est de créer un site web hébergé sur un serveur. Pour créer ce site web, les développeurs ont, à leur disposition, un SDK qui leur permet de décrire l'interface utilisateur ainsi que d'accéder aux fonctionnalités des smartphones à travers des APIs JavaScript. Contrairement aux sites web classiques qui sont habituellement interprétés à travers des navigateurs web tels que Google chrome, Safari, etc. ici, l'interprétation se fait à travers une application tierce nommée "Reload client" sur mobile et le navigateur web "UI Client". La particularité de ces applications est leur interprétation des APIs de MoSync. Par exemple, pour les APIs liées à l'interface utilisateur, l'application "Reload client" va transformer une interface web en une interface ayant un aspect natif. Ainsi, lorsque le site web est chargé sur le smarpthone, il a un aspect natif en concordance avec la plate-forme cible contrairement aux sites web classiques. L'objectif du web classique est principalement d'unifier le développement et l'interprétation des langages web. Dans le cas de MoSync, ils unifient uniquement le développement.

Ces deux dernières solutions impliquent que les utilisateurs installent une application ou un service tiers sur leur téléphone. Avec l'expérience que nous avons acquise à Keyneosoft, il est souvent très difficile de faire accepter cette manipulation aux utilisateurs. De plus, malgré ces nouvelles fonctionnalités, certaines fournies par le développement natif ne pourront jamais être implémentées à travers un site web :

— Un mode 100% déconnecté. Il n'est pas rare dans les applications mobiles de fournir un mode déconnecté. Dans ce cas, nous téléchargeons les données nécessaires au fonctionnement de l'application lors de la première utilisation. Ensuite, l'utilisateur peut naviguer dans son application sans internet. Il est possible d'intégrer ce genre de processus sur un site web. Cependant, le navigateur limite le volume possible de données à enregistrer.

— Le téléchargement de contenus supplémentaires. Par exemple, les applications liées aux chaines de télévision permettent à leur utilisateur de télécharger des épisodes de leurs séries préférées. Les utilisateurs peuvent ensuite regarder les épisodes téléchargés sans internet. Cette fonctionnalité ne peut être implémentée à partir d'un navigateur

16. Présentation officielle de MoSync reload : http ://www.mosync.com/docs/reload/guides/quick-start/mosync-reload-quick-start/index.html

internet.
- Les notifications pushs. Elles servent habituellement à prévenir des nouveautés disponibles dans une application. Nous avons, par exemple, à Keyneosoft implémenté des applications de lecture de catalogues produits pour plusieurs enseignes. À chaque fois, nous avons implémenté les notifications pushs pour prévenir l'utilisateur que des nouveaux catalogues étaient disponibles.
- L'enregistrement des données utilisateurs sur le cloud.
- Une visibilité sur les magasins d'applications officiels (play store, app store etc.)
- etc.

Aujourd'hui, toutes ces fonctionnalités nous permettent de fournir une expérience utilisateur que les utilisateurs ne trouveront pas ailleurs. Cela s'explique par le fait qu'une application mobile lorsqu'elle est installée sur un mobile est indépendante des autres applications. Elle a son propre identifiant, elle a son propre espace disque etc. Cela ne pourra jamais se faire par l'intermédiaire d'un navigateur web. L'accès aux fonctionnalités matérielles des smartphones ne suffit donc pas à fournir une expérience utilisateur égale au développement natif.

Les applications web encapsulées

Pour fournir ces fonctionnalités propres aux applications natives, une nouvelle approche a pour objectif d'encapsuler un site web à l'intérieur d'une application mobile, figure 3.5. Les SDKs natifs fournissent, en effet, le moyen de charger un site web local dans un composant graphique web : une WebView. Cordova/PhoneGap [Dav12, REUKH13, War14] est précurseur dans cette nouvelle approche. Cette technologie permet de créer un site web local à une application qui a la possibilité d'accéder aux fonctionnalités fournies par le système d'exploitation hôte telles que la caméra, accéléromètre, contacts, etc. PhoneGap permet aussi d'accéder aux fonctionnalités fournies par le système d'application mobile telles que le stockage de données, l'accès à une base de données, etc. Pour ce faire, PhoneGap est fourni avec une liste d'APIs JavaScript qui peuvent être intégrées dans n'importe quel script JavaScript. Cependant, ces APIs sont limitées à la partie non visible de l'application.

Pour rendre un aspect mobile à l'interface utilisateur des applications, il faut donc intégrer d'autres frameworks. PhoneGap est compatible avec plusieurs frameworks JavaScript [Kos12] tels que JQuery mobile [17] [Doy14], Sencha Touch [18] etc. Ces deux frameworks permettent, par exemple, de fournir à un site web un aspect mobile natif (bouton, checkbox, liste, etc.) avec des interactions utilisateurs qui s'inspirent fortement de ce que le développement natif est capable de fournir (double tap, zoom, etc.). **QuickConnectFamily** [19] [Bar09] et **Rhomobile Suite** [20] [Nal11] suivent le même principe. **MoSync** [21] a aussi fourni le moyen d'embarquer le site web dans une application native en plus de pouvoir l'héberger sur un serveur.

Précédemment, nous avons cité quelques manques liés aux applications mobiles natives. Parmi les manques cités, ces solutions permettent d'exécuter une application web sans internet. Le site web est intégré à l'application. Il n'y a donc pas besoin de communiquer avec un serveur tiers. Le téléchargement de contenus supplémentaires peut aussi être intégré dans ce genre d'application. Les développeurs auront accès à un espace de stockage comme toutes les applications natives. Cependant, certaines fonctionnalités comme la gestion des données sur

17. Démonstrations des APIs fournies par JQuery Mobile : http ://demos.jquerymobile.com/1.4.4/
18. Démonstrations des APIs fournies par Sencha Touch : http ://cdn.sencha.com/touch/sencha-touch-2.3.1a/built-examples/kitchensink/index.html
19. Site officiel de QuickConnectFamily : http ://www.quickconnectfamily.org/qc_hybrid/
20. Site officiel de Rhomobile Suite : http ://rhomobile.com
21. Création d'une application web embarquée avec MoSync : http ://www.mosync.com/docs/sdk/js/guides/quick-start/getting-started-html5-and-javascript/index.html

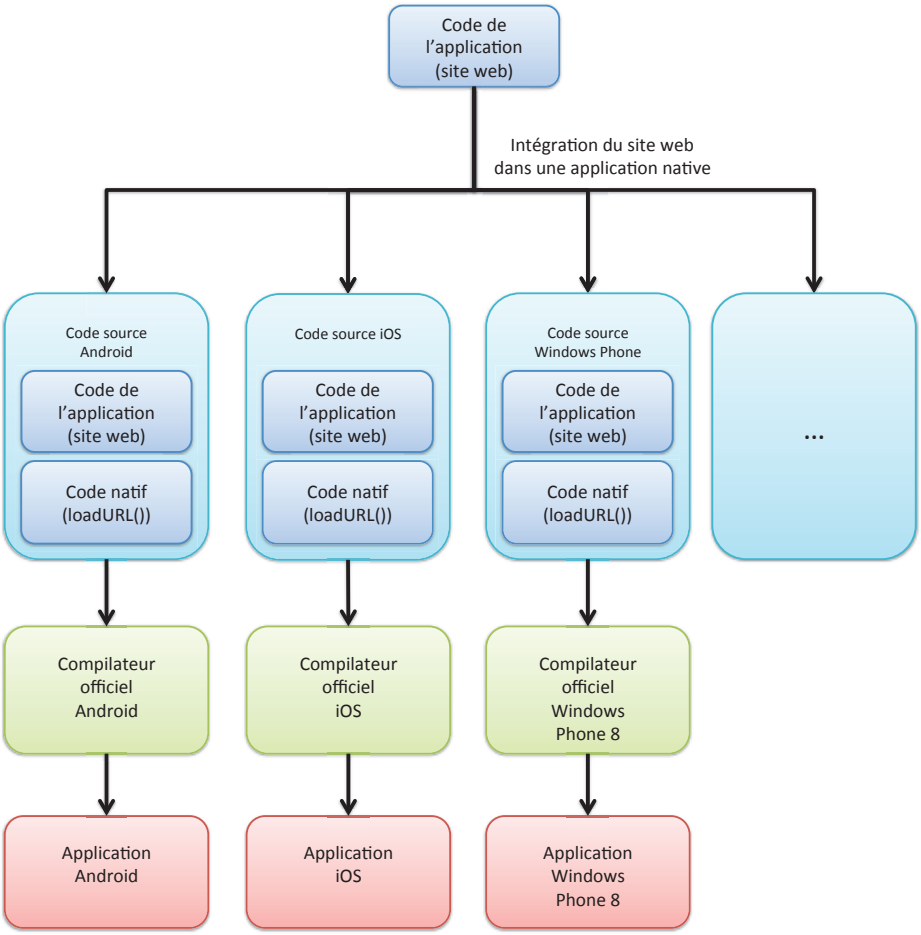


FIGURE 3.5 – Principe de l'encapsultation d'un site web dans une application mobile

le cloud ou encore les notifications pushs ne sont pas du tout traitées par ces frameworks.

Pour pallier à ces manques, **Telerik Platform** [22] permet d'utiliser PhoneGap pour développer la base d'une application web encapsulée et d'ajouter en plus toutes les fonctionnalités manquantes à PhoneGap telles que la gestion du cloud ou des pushs par exemple.

Malgré le fait qu'il est possible de fournir les mêmes fonctionnalités qu'une application mobile, ce genre de solution fournit des performances moins intéressantes que le développement natif [CSS12, Per11]. Nous complétons les études déjà menées sur PhoneGap dans le chapitre 10. Pour cela, nous avons implémenté une application entièrement avec PhoneGap et nous avons ensuite montré que les performances, par rapport au natif, sont moins intéressantes. Les mauvaises performances de PhoneGap ne sont pas uniquement visibles sur les processus qui tournent en arrière-plan (accès aux bases de données, web services, etc.) mais se ressentent aussi sur l'expérience utilisateur qui est directement perçue par l'utilisateur final. Cela s'explique principalement par le fait qu'un interpréteur de code javascript s'occupe de traiter les appels aux APIs de PhoneGap et que la partie graphique est traitée par une WebView. En effet, les WebViews ne sont pas aussi optimisées que les navigateurs web. Elles fournissent, par conséquent, de moins bonnes performances.

Titanium mobile [Bro13] permet d'implémenter une application mobile pour Android, iOS et BlackBerry. Le développement de l'application est divisé en deux partie distinctes :

— La définition de l'interface utilisateur en XML et la gestion des évènements utilisateurs.

22. Site web officiel de Telerik Platform : http ://www.telerik.com/platform#overview

— La logique applicative et l'accès aux données en JavaScript.

Pour ce faire, Titanium mobile fournit un SDK suivant le design pattern MVC. Ce SDK contient plusieurs APIs qui sont, en théorie, toutes communes aux trois plates-formes ciblées. Pour l'accès au fichier, il y aura donc uniquement une seule API à utiliser que l'on soit sur Android, iOS ou BlackBerry. En réalité, en fonction des plates-formes cibles, les APIs de Titanium mobile peuvent se comporter différemment. Contrairement à PhoneGap, les applications implémentées avec cet outil ne sont pas considérées comme des sites web mais comme des applications web mobiles. Elles se comportent de la même façon en matière d'ergonomie : une page par vue. Sur PhoneGap, il n'y a qu'une vue qui est rechargée en fonction des pages web à afficher. De plus, toutes les interactions utilisateurs (double tap etc.) sont disponibles dans le SDK de Titanium mobile. Il n'y a donc pas besoin d'utiliser un autre framework existant pour cela. Enfin, lorsqu'il est nécessaire, il est possible d'intégrer du code source natif en plus du code javascript. L'outil s'occupe alors de créer des proxys entre le code natif et le code JavaScript. Ainsi, à partir du code source JavaScript, nous pouvons appeler du code natif. Cette fonctionnalité est très utile pour intégrer du code source déjà implémenté nativement. Le code source devra bien sûr être adapté pour Titanium mobile.

Pour éviter les problèmes de performances que rencontre PhoneGap, Titanium mobile n'utilise pas les WebViews lors de l'exécution de l'application. Cette technologie utilise, en effet, son propre moteur d'affichage basé sur un interpréteur javascript : Rhino [23] de Mozilla pour Android et BlackBerry et Javascriptcore d'Apple pour iOS. JavaScriptCore est notamment utilisé dans WebKit de Safari. Pour améliorer encore les performances [24], sur Android, Titanium mobile a couplé Rhino avec V8 [25]. V8 est un interpréteur de JavaScript écrit par Google. Il est notamment utilisé par Google Chrome. Grâce à ces différents interpréteurs de JavaScript, Titanium mobile arrive à augmenter considérablement ses performances. Cela se ressent au niveau de l'utilisation des applications générées. L'expérience utilisateur est bien meilleure qu'avec PhoneGap.

Malgré des performances raisonnables, l'outil d'interprétation du code source JavaScript n'est pas fiable. Il plante régulièrement sur certains téléphones alors que sur d'autres, il n'y a aucun problème. Il est difficile pour une entreprise de tester sur tous les smartphones possibles. Il se pourrait donc en déployant une application Titanium sur les markets officiels (google play, app store ...) qu'elle ne fonctionne pas sur une partie des smartphones sur lesquels elle sera installée. Dans ce cas, l'application sera mal notée et ne sera plus téléchargée. Il est donc très difficile d'utiliser Titanium mobile dans un cadre professionnel. De plus, les outils de développement utilisés sont, eux, très difficiles à appréhender. Ils ne fournissent, par exemple, pas d'outil d'analyse syntaxique ce qui permettrait de détecter rapidement les erreurs liées à l'écriture du code. Il faut souvent attendre d'exécuter l'application pour se rendre compte qu'une méthode appelée n'existe pas, par exemple. Pour finir, comme nous l'avons dit précédemment, en fonction de la plate-forme cible, les APIs de Titanium mobile ne se comportent pas de la même façon. Ces différences de comportement sont très difficiles à gérer pour les développeurs. Cela les oblige à tester régulièrement leurs applications sur toutes les plates-formes cibles de leurs applications. Le gain pendant la phase de développement grâce à un code source commun diminue alors avec les tests à répétition que les développeurs doivent effectuer. Nous détaillons tous ces points dans le chapitre 10 où nous évaluerons plus précisément Titanium mobile.

23. Site officiel du moteur de JavaScript : https ://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

24. Ticket sur l'amélioration des performances de Titanium mobile : http ://www.appcelerator.com/blog/2011/09/platform-engineering-android-runtime-performance-improvements/

25. Site officiel du projet V8 de Google : https ://code.google.com/p/v8/

Les widgets mobiles

Les **widgets mobiles** sont des sites web locaux au smartphone implémentés avec des technologies webs (notamment avec le HTML 5). Contrairement à un site web classique ou les solutions présentées précédement, ils sont dédiés au développement mobile et ne fournissent qu'une fonctionnalité (gestion de la batterie, calendrier, réveil, etc.). De base, ils prennent en compte tous les aspects du domaine du mobile (taille d'écran. etc.). Pour les exécuter, un moteur de widgets est installé sur le smartphone et s'occupe de faire le lien entre le widget et la plate-forme cible. C'est le moteur d'exécution qui gère l'aspect multiplateforme du widget. Comme pour Titanium mobile, l'objectif est de ne pas utiliser les WebViews comme support d'exécution mais de concevoir un moteur d'exécution optimisé pour les widgets.

La spécification des widgets mobiles a été lancée par la **Wholesale Applications Community** [26] (WAC). C'était une alliance de plusieurs sociétés de télécommunication. Parmi ces sociétés, certaines étaient spécialisées dans le développement du moteur d'exécution des widgets. Par exemple, **Borqs** [27] a fourni un moteur de widgets compatible avec les spécifications de la WAC pour OPhone OS, Symbian et Windows Mobiles. Il y a eu environ 24000 widgets WAC qui ont vu le jour pour ce moteur. **Obigo** [28] a effectué le même travail pour BREW et windows mobile alors qu'**Opéra** [29] s'est concentré sur Android [30] Windows Mobile et Symbian OS. Suite à des conflits entre les acteurs de la WAC, le travail effectué sur les widgets n'a pas abouti sur un déploiement commercial. Cependant, le WAC a sorti une spécification sur les widgets [31]. Le packaging des widgets WAC est basé sur le travail du W3C [32] [Các12]. Aujourd'hui, plus aucun moteur de widgets n'est maintenu pour Android et aucun n'a été crée pour iOS ou Windows Phone.

Selon [JFL10], les moteurs de widgets sont généralement lourds. Pour éviter cela, dans [JFL10], les auteurs présentent **xFace** un moteur de widgets léger et multiplateforme implémenté en C++. Il a implémenté pour Android, Windows Mobile, Symbian S60, MTK, Brew, Moblin. Ce moteur est léger car il ne prend en compte que les besoins liés au domaine du mobile. Par exemple, toutes les balises HTML qui ne peuvent pas être utilisées dans le cas d'une application mobile sont exclues du moteur. Ainsi, le nombre de règles implémentées est plus restreint. De plus, pour porter facilement leur moteur de widgets sur d'autres plates-formes, ils l'ont conçu à partir d'une architecture à composants [PXL10]. La couche de portage est divisée en plusieurs composants (multémédia, gestion des fichiers...). Il suffit, alors, de faire le lien entre chaque composant et chaque plate-forme. Par exemple, pour faire le lien entre xFace (C++) et Android (Java), ils ont utilisé JNI [33] [WLL10].

Pour finir, les auteurs de [CR10] présentent un générateur automatique de widgets mobiles. Ils ont conçu un langage de haut niveau d'abstraction basé sur XML : le **Widget Markup Language**. Le développeur définit alors son widget et ses fonctionnalités grâce à ce langage. Le compilateur génère ensuite ce widget pour plusieurs moteurs de widgets. Cette méthode permet alors, de fournir un widget pour plusieurs plates-formes mobiles.

26. Description du projet de Wholesale Applications Community : http ://www.visionmobile.com/blog/2010/04/wholesale-applications-community-the-operator-love-affair-with-developers/
27. Site officiel de Borqs sur les widgets mobiles : http ://www.borqs.com/products.aspx ?ProductsCateID=419&CateID=416&Curr
28. Site web officiel d'Obigo : http ://www.obigo.com/main.ob ?page=7
29. Site web officiel d'Opéra : https ://dev.opera.com
30. Moteur de widget Opéra pour Android : https ://dev.opera.com/blog/widget-runtime-wac-1.0-golden-for-android/
31. Spécification WAC sur les widgets mobiles : http ://specs.wacapps.net/1.0/dec2010/overview.html
32. Liste des spécifications du W3C sur les widgets : http ://www.w3.org/2008/webapps/wiki/WidgetSpecs
33. JNI : Java Native Interface. JNI permet de faire le lien entre des méthodes s'exécutant sur une machine virtuelles Java et une application écrite dans un langage différent.

Cette solution ne répond pas entièrement à nos besoins. En effet, les widgets ne permettent que d'implémenter de petites applications avec une ou deux fonctionnalités. De plus, plusieurs fournisseurs de moteurs de widgets sont présents sur le marché. Le risque est d'avoir les mêmes problèmes que les navigateurs web. Les développeurs devront pour chaque moteur de widget adapter leurs widgets. Enfin, il serait possible que plusieurs moteurs de widgets soient installés sur le même smartphone. Aujourd'hui, rien n'est prévu pour faire face à ce cas.

3.4 Conclusion

Pour rappel, nous voulons une solution qui permette de développer une application mobile multiplateforme le plus rapidement possible. Pour cela, il faut cacher ou atténuer les différences entre les plates-formes mobiles. Pour autant, ces solutions ne doivent pas fournir de limitation au développement mobile, c'est-à-dire, que les applications générées doivent avoir un aspect natif (propre à chaque plate-forme) et fournir les mêmes fonctionnalités que le développement natif (accès aux capteurs, paiement in app, push, etc.). De plus, ces solutions doivent être suffisamment flexibles pour prendre en compte les évolutions rapides du domaine du mobile.

Dans un premier temps, nous avons étudié les solutions permettant le développement multiplateforme qui sont proches des systèmes d'exploitation mobiles. Cependant, aucune ne répond à tous nos besoins.

— Les compilateurs source à source permettent d'implémenter une seule fois une application dans un langage natif. Cependant, ils sont souvent limités aux fonctionnalités communes de chaque plate-forme. Dès qu'un développeur veut intégrer une fonctionnalité propre à la plate-forme, la maintenance de son application sera difficile. De plus, ce genre de solutions est difficile à mettre en place. Il faut, en effet, faire le lien entre les librairies du langage commun et les librairies de chaque plate-forme cible, ce qui est un travail considérable. Le plus difficile est de faire évoluer ce genre d'outil avec les évolutions du domaine du mobile.

— Les solutions basées sur le web permettent maintenant d'accéder aux fonctionnalités natives des smartphones que ce soit à partir d'un navigateur web ou dans une application tierce qui fait office d'interpréteur d'un site web local. Cependant, ces solutions ne permettent pas de fournir une expérience utilisateur aussi riche que le développement natif. L'interface utilisateur ressemble plus à un site web qu'à une application native et les performances sont moins intéressantes.

Pour aller plus loin, nous avons fait une comparaison plus précise de PhoneGap, Titanium mobile et Xamarin dans le chapitre 10.

Dans un second temps, nous avons étudié les solutions qui font complètement abstraction des environnements mobiles existants. C'est l'objet du prochain chapitre.

Chapitre 4

Modèle de développement prenant en compte l'hétérogénéité des systèmes

D'autres solutions permettent de développer une application mobile une seule fois et ensuite de la déployer pour n'importe quelle plate-forme mobile. Dans ce chapitre, nous nous focalisons sur les solutions de haut niveau, c'est-à-dire que ces solutions ne se basent pas sur les plates-formes existantes pour fournir un environnement de développement unique. L'objectif est plutôt de définir un environnement de développement pour le domaine du mobile et non pour ce qui existe aujourd'hui. Après avoir passé en revue ce genre de solution, nous étudions celles qui fournissent un environnement d'exécution mobile unique. Dans ce cas, les développeurs implémentent leurs applications pour un environnement d'exécution qui fonctionne sur n'importe quelle plate-forme. Nous terminons ce chapitre, avec une présentation de la programmation par composants qui permet notamment de diminuer le temps de développement d'une application. C'est un des points que nous aimerions trouver dans une solution pour le développement multiplateforme mobile.

Sommaire

4.1 Introduction 44
4.2 Les langages spécifiques à un domaine (DSL) et l'ingénierie dirigée par les modèles (IDM) 44
4.2.1 Les DSLs : Domain Specific Language 44
4.2.2 Ingénierie dirigée par les modèles (IDM) 45
4.2.3 Discussions 48
4.3 Environnement d'exécution commun 48
4.3.1 Machine virtuelle 48
4.3.2 Délégation des services sur des serveurs/cloud 50
4.3.3 Portage d'environnements mobiles 51
4.4 La programmation par composants 52
4.5 Conclusion 56

4.1 Introduction

Dans le chapitre précédent, nous nous sommes focalisés sur les approches permettant le développement d'applications mobiles multiplateformes à partir de solutions proches des systèmes d'exploitations et SDKs natifs. Nous pouvons même qualifier ces solutions de bas niveau ou plutôt proches des systèmes d'exploitation existants. Leur objectif est de fournir un environnement de développement qui réunit les SDKs existants à travers un seul SDK qui lui est commun à toutes les plates-formes cibles. Pour la définition du SDK commun, les concepteurs se basent sur les SDKs natifs de chaque plate-forme.

Dans ce chapitre, section 4.2, nous nous focalisons dans un premier temps sur les solutions de plus haut niveau. Contrairement aux solutions déjà présentées, l'objectif n'est pas de fournir un environnement de développement pour les plates-formes existantes mais un environnement permettant de réaliser des applications pour le domaine du mobile. Ensuite, section 4.3, nous présentons des solutions qui fournissent un environnement d'exécution commun à plusieurs plates-formes mobiles. L'objectif n'est plus de générer une application native par plate-forme mais d'exécuter la même application sur toutes les plates-formes mobiles. Enfin, section 4.4, nous terminons avec une présentation de la programmation par composants qui a la particularité d'accroître la productivité des développeurs. Nous nous baserons sur ce type de programmation pour proposer une alternative aux solutions existantes.

4.2 Les langages spécifiques à un domaine (DSL) et l'ingénierie dirigée par les modèles (IDM)

Dans les approches présentées dans le chapitre précédent, section 3.3.1, le principal objectif était de mutualiser les SDKs natifs de chaque plate-forme cible sans pour autant s'abstraire de toutes les spécificités de chacun des environnements. Dans les solutions basées sur des DSL (Domain Specific Language) et l'IDM, l'objectif est plutôt de construire un environnement de développement pour un domaine particulier, ici celui du mobile, sans prendre en compte les spécificités de chaque SDK natif. Ainsi les développeurs se focalisent uniquement sur des fonctionnalités de haut niveau et à l'assemblage de ces fonctionnalités.

4.2.1 Les DSLs : Domain Specific Language

Les DSLs sont des langages communs conçus indépendamment d'un système d'exploitation particulier et pour le domaine du mobile. Pour ce faire, avant de créer un DSL, il faut étudier le domaine pour faire ressortir les aspects essentiels que l'on attend d'une application [KCO11, LGW13]. Par exemple, pour le domaine du mobile, les développeurs ont besoin de gérer plusieurs tailles d'écrans, de développer leurs interfaces graphiques sous la forme de piles (piles de vues), les interactions utilisateurs dirigent en quelque sorte le développement etc. Tous ces aspects sont alors pris en compte et intégrés dans un langage commun. Contrairement aux langages "génériques", les langages spécifiques à un domaine ne peuvent être utilisés que pour créer des applications spécifiques au domaine. Ils sont définis uniquement pour cela. Lorsque le langage est défini, un compilateur est fourni pour chacune des plates-formes cibles du langage. Ces compilateurs se classent dans la même catégorie que les compilateurs sources à sources présentés dans le chapitre précédent, section 3.3.1.

Le langage de programmation **MobDSL** [KCO11] a été conçu en suivant cette méthode. Uniquement après que le langage ait été spécifié, les liens entre lui et les librairies natives ont

été implémentés. Dans le cas de MobDSL, le langage est interprété par un moteur capable de faire le lien entre les systèmes d'exploitation cibles et l'application. Pour cela, il se base sur des librairies compatibles avec chaque système d'exploitation.

Dans le même ordre d'idées, **Xmob** [LGW13] a été conçu en suivant le design pattern MVC. Le langage est donc divisé en trois parties : xmob-data, xmob-ui et xmob-event. Pour transformer le code source écrit avec Xmob pour plusieurs plates-formes, ils ont basé leur processus sur l'ingénierie dirigée par les modèles. Nous revenons sur ce système dans la section 4.2.2.

Le framework **DIMAG** [MMOR09] permet de générer une application par plate-forme cible à partir d'un langage commun déclaratif. Les applications écrites avec DIMAG sont divisées en trois parties. La première est implémentée avec le langage DIMAG-root qui permet de décrire le workflow de l'application, DIMAG-UI qui permet de définir l'interface utilisateur et une partie implémentée en SCXML (State Chart eXtensible Markup Language) qui permet de définir tous les états de l'application ainsi que les conditions permettant de passer d'un état à un autre. Les applications générées avec cette solution sont exécutées à travers une machine virtuelle.

Mobl [HV11] a été conçu de la même façon. Contrairement à MobDSL, ce langage permet de faire des applications web. Il est divisé en deux parties distinctes : la partie interface et la partie données enregistrées localement dans le navigateur. Pour chacun de ces deux aspects des applications mobiles, un DSL a été conçu. Ces deux DSLs ont la particularité d'être uniquement déclaratif. Les applications écrites avec ce langage sont entièrement transformées en JavaScript et HTML5 grâce à un compilateur source à source. Elles sont donc exécutées à travers des navigateurs web installés sur les appareils mobiles. Les concepteurs de ce langage ont également défini un moyen de transformer rapidement un DSL en code source cible [HV09]. Ils ont introduit un langage intermédiaire entre le DSL et le code source cible : PIL (Platform Independant Language). Ce langage intermédiaire est moins riche que le DSL ce qui simplifie la mise en place des liens avec langage cible. L'objectif est de faciliter l'ajout de plates-formes cibles au DSL. Dans la plupart des solutions présentées précédemment, la transformation du DSL en langage natif passe par un compilateur spécifique à chaque plate-forme. La spécification de ce compilateur est plus ou moins compliquée en fonction des différences entre le DSL et le langage cible. Chaque fois qu'une nouvelle plate-forme est ajoutée, il faut, bien sûr, re-développer le compilateur. En ajoutant un langage intermédiaire comme PIL, cette tâche est alors moins difficile.

4.2.2 Ingénierie dirigée par les modèles (IDM)

L'ingénierie dirigée par les modèles est un paradigme de programmation apparu dans les années 2000. Cette approche a été initialement proposée par le group Object Managment Group [1] à travers MDA [BCW12] (Model Driven Architecture). Contrairement au développement classique, avec cette approche, le développeur n'écrit pas ou très peu de lignes de code. Le principe est de décrire leurs applications à l'aide de modèles. Les modèles sont basés sur un métamodèle qui représente en quelque sorte la grammaire que les développeurs peuvent utiliser et doivent respecter. Pour notre problématique, ces modèles sont indépendants d'une plate-forme spécifique et définis pour le domaine du mobile [Cha13]. Ces modèles décrits sont ensuite transformés en code natif pour chacune des plates-formes cibles grâce à des outils de transformations et de génération de codes [Fav06]. Nous avons représenté ce processus adapté à notre problématique sur les figures 4.1a et 4.1b. Dans notre cas, la com-

1. http ://www.omg.org/

pilation du code source générée en application finale se fait avec les compilateurs officiels de chaque plate-forme. Plusieurs travaux ont déjà montré que cette approche basée sur la modélisation permet de simplifier le développement mobile pour une plate-forme spécifique [GBCP14, Kra11, BE08, MKS+11].

Deux façons de transformer les modèles peuvent être exploitées. Sur la figure 4.1a, il est possible de passer directement des modèles de l'application aux différents codes sources natifs. Alors que sur la figure 4.1b, une étape supplémentaire est ajoutée. Dans ce cas, les modèles indépendants d'une plate-forme cible sont transformés dans un premier temps en d'autres modèles spécifiques à chaque plate-forme cible. Enfin, ces nouveaux modèles sont transformés en code source natif.

Dans ce genre d'approche, la modélisation du domaine est très importante. L'approche est d'ailleurs la même que pour la conception d'un DSL. Les concepteurs doivent identifier les besoins du domaine et s'abstraire des SDKs classiques. D'ailleurs, les DSLs peuvent être combiné avec l'IDM comme Xmob [LGW13], par exemple. Avec Xmob, les développeurs implémentent donc leurs applications à partir d'un langage de programmation commun. Le code écrit est ensuite transformé sous forme de modèle dépendant à chaque plate-forme cible, figure 4.1b. Ce sont ces modèles qui sont ensuite transformés en langage natif.

Capucine [PQD12] permet de décrire des applications mobiles à travers une SPL (Software Product Line). Dans ces travaux, les développeurs se basent sur le métamodèle d'AppliDE [QDDD11]. Il est composé de trois parties :

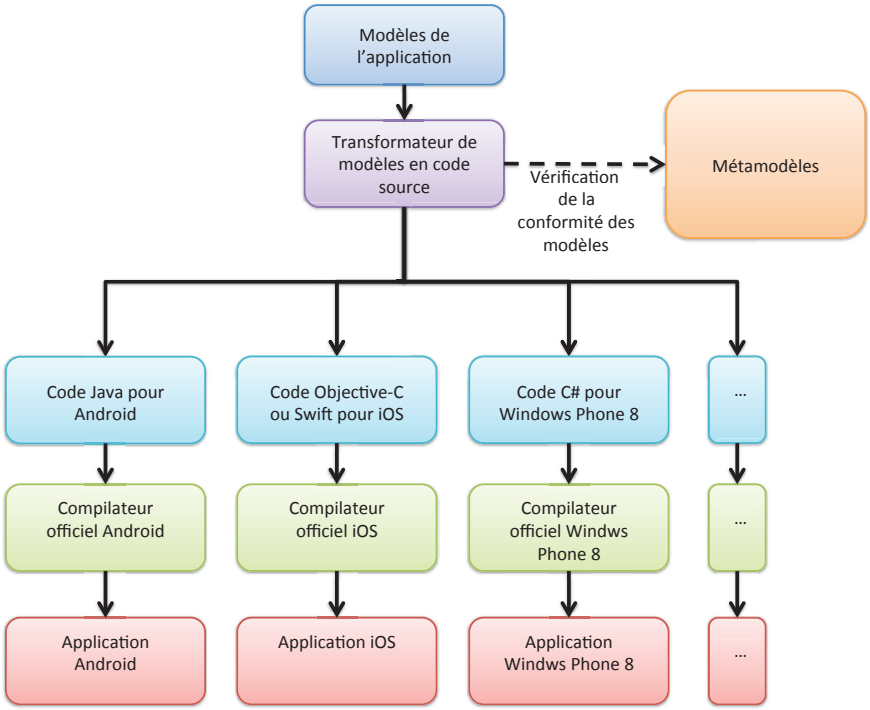
— Les fonctionnalités souhaitées (envoi de mail, GPS, etc.).
— Les différentes sources de données (fichier, web services, bases de données, etc.).
— Les éléments graphiques (listes, boutons, champs textes, etc.).

Avec ce métamodèle, les développeurs peuvent décrire leurs applications avec des diagrammes de features [KCH+90, PBL05] . C'est grâce aux diagrammes de features que les développeurs décrivent le comportement de leurs applications [PCBD10]. En effet, pour chaque cas possible (pas de connexion internet, etc.), les développeurs devront choisir entre telle ou telle fonction (définie à partir des éléments du métamodèle). Ensuite, par transformation, les différents modèles de features sont transformés en code natif pour chacune des plates-formes cibles [QMPD11].

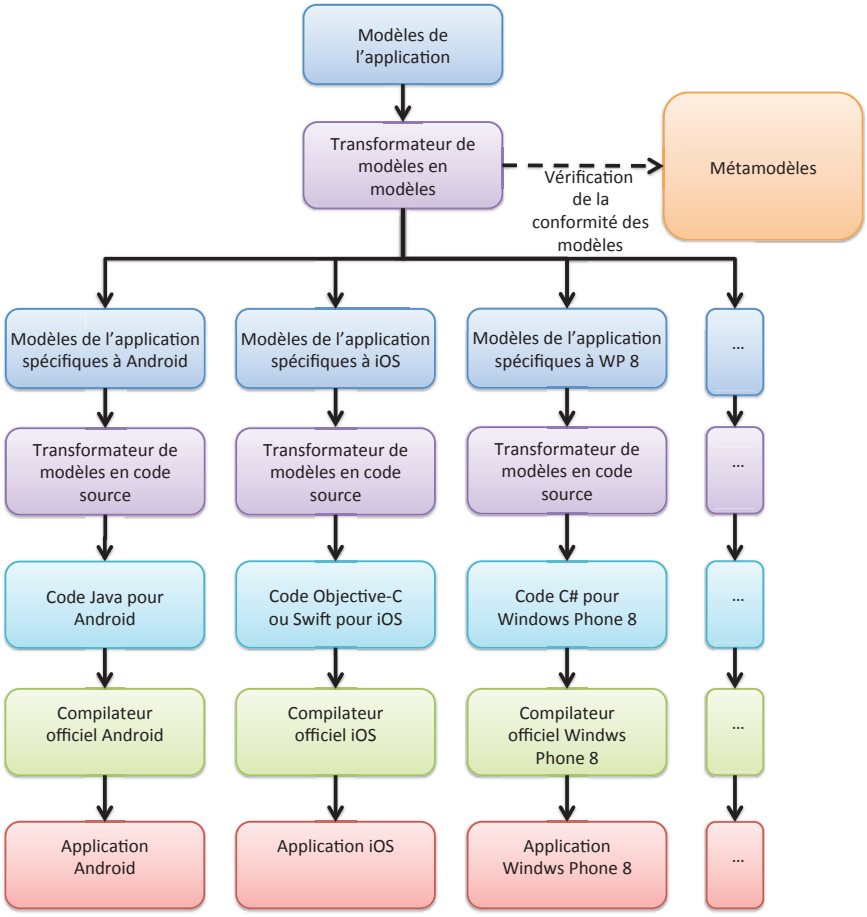
Dans le même ordre d'idées, dans [AGPK11] les auteurs ont conçu un outil permettant de décrire une application à l'aide du langage de modélisation EP [Kel06, KM08] et des templates FTL (Formal Template Language) [ASP06]. Comme pour la solution précédente, les modèles sont transformés pour chaque plate-forme cible.

La conception de **Mobia modeler** [BFH08] est également basée sur cette approche. Pour simplifier le développement des applications, ils ont combiné l'ingénierie dirigée par les modèles avec des techniques centrées utilisateurs dans le domaine des interfaces hommes-machines. Leur objectif est de cibler les non-informaticiens. Pour cela, ils ont créé un langage de modélisation graphique spécifique au domaine des mobiles et indépendant des plates-formes mobiles : Modia modeler [BFTH10]. Le processus de compilation se passe de la façon suivante : le modèle est transformé en XML Schéma. Ce fichier XML est exporté sous le format Mobia PIM (Platform Independant Model), et enfin le code est généré pour une plate-forme cible à l'aide de templates implémentés spécialement pour chaque plate-forme cible.

Les modèles peuvent être décrits de façon graphique à l'aide de diagrammes UML (Unified Modeling Language). C'est le cas de **XIS-mobile** [RdS14, dS03], par exemple. Cependant, parfois, les modèles sont définis à partir de modèles décrits textuellement. C'est le cas de md^2 [HMK13] qui est un langage de déclaration de modèle textuel pour le développement mobile multiplateforme.



(a) Transformations de modèles indépendants de toutes les plates-formes cibles en code source natif



(b) Transformations de modèles indépendants de toutes les plates-formes cibles en modèle spécifique à chaque plate-forme visée puis en code source natif

FIGURE 4.1 – Principe de l'ingénierie dirigée par les modèles appliquées au développement d'applications mobiles

4.2.3 Discussions

Contrairement aux solutions présentées dans le chapitre précédent, l'objectif est de s'abstraire de la couche de bas niveau des applications. Plus le niveau d'abstraction est élevé, plus le développement multiplateforme est facile à mettre en place. Cependant, en faisant abstraction d'un certain nombre de fonctionnalités de bas niveau, il se peut que ces solutions limitent le développement de certaines parties de ces applications. Il n'est, en effet, pas rare que dans nos applications chez Keyneosoft, nous utilisions des fonctionnalités de bas niveau pour répondre à un besoin particulier. Par exemple, avec ce genre de solution, les fonctionnalités liées à la caméra sont souvent "prendre une photo" ou "prendre une vidéo" alors que parfois nous avons plutôt besoin de récupérer le flux vidéo pour faire des traitements particuliers. Avec ce genre de solution, ce sera très difficile de le mettre en place. Dès que nous souhaitons fournir des fonctionnalités très fines, ce sera donc difficile de l'implémenter.

Pour nous, le fait de se concentrer uniquement sur le domaine du mobile et de ses fonctionnalités sans se soucier des implémentations natives pourrait nous limiter dans les fonctionnalités à réaliser. De plus, en utilisant ce genre d'outil, nous serions complètement dépendant de la solution. Si une nouvelle fonctionnalité sort sur l'un des systèmes d'exploitation, elle ne sera accessible que lorsque les concepteurs de la solution l'exposeront.

Ensuite, aujourd'hui, aucune évaluation de ces outils n'a été effectuée dans un cadre professionnel. Pour valider la mise en place, il faudrait implémenter une application professionnelle, c'est-à-dire, une application qui pourrait être déployée sur les magasins d'applications officiels. Le but serait, bien-sûr, de savoir combien de temps les développeurs pourraient gagner en les utilisant. En complément, il serait intéressant d'étudier la taille des modèles nécessaires à la réalisation d'une telle application. Chez Keyneosoft, les applications développées dépassent souvent les 30000 lignes de code. Est-ce que les modèles pour les réaliser ne seraient pas trop complexes ? Nous pouvons quand même facilement imaginer que les modèles pour décrire nos applications seraient très volumineux. Est-ce qu'à partir d'une certaine taille de modèles, les modèles sont encore exploitables ? Quels sont les outils de débogage utilisés (breakpoint, console, etc.) ? Aujourd'hui, les études menées ne nous permettent pas encore d'avoir ces réponses.

Dans les perspectives de cette thèse, une des pistes sera de combiner ce genre d'approche, qui permet notamment d'assembler des fonctionnalités de haut niveau, avec notre proposition qui est, elle, très proche des systèmes d'exploitation natifs.

4.3 Environnement d'exécution commun

Dans cette section, nous nous focalisons sur les solutions ayant comme objectif de fournir un environnement d'exécution commun aux applications mobiles. Le principe est d'installer sur tous les smartphones l'environnement d'exécution et ensuite de développer des applications uniquement pour cet environnement. Ainsi, les développeurs implémentent une seule fois leurs applications.

4.3.1 Machine virtuelle

Sur les ordinateurs classiques, aujourd'hui, le Java permet de faire des applications multiplateforme pour Windows, Linux, Mac OS X etc. Pour ce faire, une machine virtuelle, la

JVM[2] [LY] est installée sur chacun des systèmes d'exploitation. C'est elle qui est chargée d'exécuter les applications Java. Contrairement à un logiciel classique, la JVM gère elle-même la consommation des ressources des applications Java en exécution. Elle s'occupe donc d'allouer des espaces mémoires pour une application, d'exécuter du code, d'ordonnancer l'exécution des applications, etc. Elle joue en quelque sorte le rôle d'un système d'exploitation classique. Sur ordinateur, ce système est très bien pris en compte depuis des années. Il a été donc naturel d'exporter ce système au mobile avec le **Java ME**[3] [Ris08, Yua03]. Cependant, à la sortie des smartphones, le concept a cessé d'être exploité par les développeurs. Même si cette solution n'est plus retenue pour le développement mobile multiplateforme, il est intéressant de comprendre pourquoi cela n'a pas fonctionné.

Java ME est composé de deux configurations CDC (Connected Device Configuration) et CLDC (Connected Limited Device Configuration). La première configuration est adaptée aux terminaux relativement puissants tels que les PDA[4]. La machine virtuelle utilisée pour ce type d'appareils est la CVM (C-Virtual Machine). Elle offre les mêmes fonctionnalités que la JVM et est optimisée pour ce type de machines. Alors que la deuxième est adaptée aux appareils à faibles capacités comme les téléphones portables, la machine utilisée pour ce type d'appareils est une KVM (Kilobyte Virtual Machine). Cette machine virtuelle n'offre pas toutes les fonctionnalités de la JVM mais est très légère (entre 40 et 80 Ko), figure 4.2.

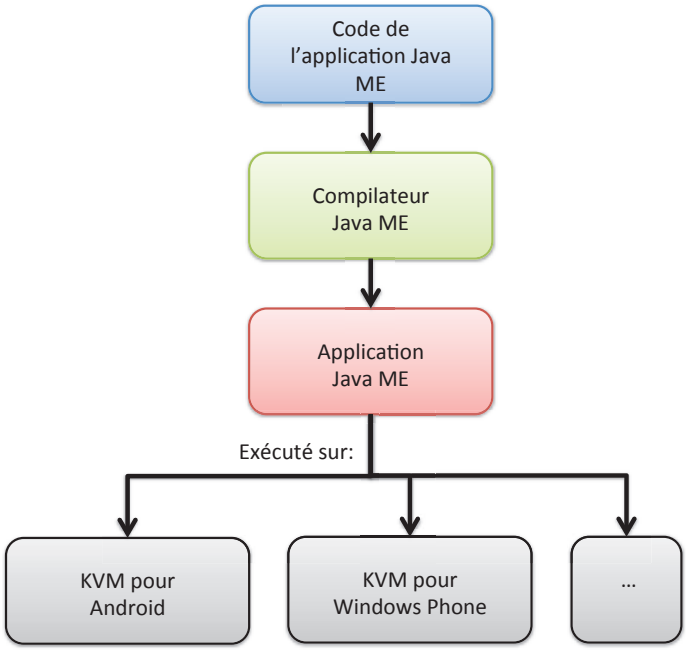


FIGURE 4.2 – Principe du développement multiplateforme avec Java ME

Java ME était principalement utilisé avant la sortie des smartphones Apple ou Android. Il était disponible pour Symbian par exemple. Il supportait un grand nombre de téléphones mais n'utilisait pas les API natives du système. Il était donc impossible d'avoir une apparence native de l'application, c'est-à-dire, qui ressemble au système d'exploitation du téléphone sur lequel l'application était exécutée. Les machines virtuelles de Java ME, surtout la KVM, étaient souvent installées sur des téléphones peu performants. L'exécution de la machine virtuelle et de l'application était donc souvent lente. De plus, l'objectif de Java ME était d'implémenter une seule fois une application pour qu'elle soit exécutée sur n'importe quel

2. JVM : Java Virtual Machine
3. Java ME : Java Micro Edition
4. PDA : Personal Digital Assistant

téléphone. La réalité était toute autre. Effectivement, pour un même système d'exploitation, il existe une multitude de JSR (Java Specification Requests) différentes selon les fonctionnalités que l'on voulait utiliser. Par exemple, pour Symbian, la JSR 82 étend MIDP [5] 2.0 permet d'utiliser le bluetooth, alors que la JSR 184 permet de créer des composants graphiques 3D. L'objectif était de fournir, en fonction des besoins, la machine virtuelle la plus adaptée. Cependant, la fragmentation entre systèmes d'exploitation était toujours présente et même encore plus prononcée avec les différentes JSR pour un même système d'exploitation [LP06]. À la sortie d'iOS et d'Android, les développeurs ont délaissé cette technologies pour développer nativement leurs applications. Dans le cas d'Android, les développeurs pouvaient installer une application sur plusieurs téléphones sans avoir besoin pour chacun des téléphones de sélectionner une JSR particulière [GHG10]. À la suite de cela, **J2ME Polish** [6] et **AlcheMo** [7] ont été conçus pour porter une application Java ME sur plusieurs systèmes d'exploitation mobiles. Le fonctionnement de ces solutions est basé sur un compilateur source à source.

Pour terminer, Apple refuse catégoriquement l'exécution d'une machine virtuelle. Ils ne veulent pas diminuer les performances de leurs applications. Cette solution ne fonctionne donc qu'au bon vouloir des fournisseurs de systèmes d'exploitation. Dans un cadre professionnel, il n'est donc pas possible d'exploiter cette solution.

4.3.2 Délégation des services sur des serveurs/cloud

Une nouvelle approche a été introduite récemment avec l'arrivée des premiers travaux sur l'économie d'énergie dans le domaine du mobile [ZJKG10]. L'objectif est d'utiliser le cloud comme une plate-forme d'exécution de certaines parties d'une application mobile [MT13]. Par exemple, une application mobile de reconnaissance d'images peut être divisée en deux parties distinctes : la capture des images avec la caméra du smartphone et la reconnaissance d'images sur le cloud. En déléguant l'exécution de processus lourds, comme dans notre exemple, sur le cloud, nous améliorons les performances de ce type de services. Effectivement, sur ce type de support, nous pouvons stocker plus de données (ici des images) et les algorithmes de reconnaissances d'images fonctionnent plus rapidement (puissance de calcul d'un serveur versus smartphone). De plus, en diminuant les calculs fait sur le smartphone, la consommation d'énergie diminue.

Pour notre problématique d'hétérogénéité ce genre de solution permet d'implémenter une seule fois certains services et d'y accéder à partir de n'importe quelle plate-forme à travers des web services, par exemple. **μcloud** [MGL+11] est conçu pour exploiter cette approche. Avec ce système, une application est divisée en plusieurs composants. Chaque composant est classifié selon sa localisation (mobile, cloud ou les deux). Bien sûr, les composants sur le cloud peuvent être réutilisés par n'importe quelle application. Enfin, l'utilisation de service provenant du cloud doit se faire de façon transparente pour les développeurs.

Ce type de solutions ne correspond qu'à une partie des applications que nous sommes amenés à implémenter. En effet, en utilisant ce genre de solution, les applications doivent être constamment connectées à Internet.

5. MIDP : Mobile Information Device Profile, c'est un ensemble d'API Java ME
6. Site web officiel de J2ME Polish : http ://www.enough.de/products/j2me-polish/
7. Site web officiel de AlcheMo : http ://www.innaworks.com/alchemo-java-me-j2me-to-brew-android-iphone-flash-windows-mobile-cross-compiler.html

4.3.3 Portage d'environnements mobiles

Une solution un peu plus radicale consiste à porter directement un système d'exploitation sur un autre système d'exploitation. Par exemple, nous pourrions porter Android sur un iPhone, ou iOS sur un smartphone Android. Ainsi, les utilisateurs auraient les deux plates-formes sur le même mobile. Ils auraient alors accès à toutes les applications de chacune des plates-formes installées. **Alien Dalvik** [8] a par exemple été installé sur des smartphones équipés de Meego et maintenant sur des smartphones Jolla équipé de Sailfish OS [9]. L'objectif est de porter la machine virtuelle Dalvik sur d'autres systèmes d'exploitation. Meego était un système d'exploitation open-source basé sur Linux. Aujourd'hui, son développement n'est plus soutenu. Il a été remplacé par le projet Tizen [10] qui a le même objectif. Comme Tizen, Sailfish OS est un dérivé du projet Meego et est installé sur les smartphones de l'entreprise Jolla [11]. **In-the-box** a le même objectif. Ils ont porté la machine virtuelle Dalvik d'Android sur iOS. Apparemment, ce projet n'est plus maintenu. Enfin, les concepteurs de **Cider** [AVHA+14] propose aussi d'installer l'environnement d'exécution d'iOS sur Android. Ainsi, les utilisateurs d'Android pourront avoir accès à toutes les applications iOS. Pour ce faire, ils ont installé le composant logiciel XNU qui s'occupe d'exécuter les applications iOS sur Android. XNU est open-source, ils ont donc pu avoir accès à ce composant.

Ce genre de solution est difficilement déployable pour le grand public. Par exemple, la solution Cider touche au Kernel Android en ajoutant une partie d'iOS dans Android. Cependant, les constructeurs et les opérateurs de téléphonie qui installent Android retouchent eux-mêmes le système d'exploitation. Ils le transforment et compilent eux-mêmes une nouvelle version d'Android en fonction de leurs besoins. Dans ce cas, Cider ne pourrait pas être installé sur ces nouvelles versions d'Android. De plus, l'installation de deux systèmes d'exploitation sur le même smartphone doit être transparente pour l'utilisateur. En effet, ce n'est pas comme sur les ordinateurs ou en fonction de nos besoins, nous passons de Windows à Linux, par exemple. L'installation d'Alien Dalvik sur les smartphones Jolla est complètement transparente. Cependant, nous pouvons nous demander quel sera l'intérêt pour les développeurs d'applications mobiles de créer des applications pour Sailfish OS. Si les applications Android sont compatibles sur ce système d'exploitation, les développeurs mobiles choisiront automatiquement Android comme cible de développement. En plus de toucher les smartphones Android, ils toucheront les smartphones Sailfish OS ce qui risque à long terme de faire perdre l'intégrité de ce système d'exploitation. Ce genre de solutions pourrait à terme unifier les systèmes d'exploitation et par la même occasion brider les innovations dans le domaine du mobile.

Aujourd'hui, dans les solutions présentées, aucune d'elle ne répond à tous nos besoins. Dans la suite, nous étudierons la programmation par composants. Même si elle n'est pas utilisée pour gérer l'hétérogénéité des terminaux mobiles, elle fournit des propriétés que nous aimerions exploiter dans le cadre de cette thèse.

8. Site web officiel d'Alien Dalvik : http ://www.myriadgroup.com/products/device-solutions/mobile-software/alien-dalvik/
9. Installation d'Alien Dalvik sur les smartphones Jolla : http ://www.jollatides.com/2014/01/31/android-on-jolla-what-kind-of-an-android-phone-do-we-have/
10. Site officiel du projet Tizen : https ://www.tizen.org/
11. Site web officiel de Jola : http ://jolla.com

4.4 La programmation par composants

Dans le chapitre 2, section 2.2, nous avons montré que le développement mobile a un coût élevé. Par conséquent, notre principal objectif est d'améliorer la productivité des développeurs lors de l'implémentation d'applications multiplateformes. Dans cette section, nous présentons donc la programmation par composant qui a la particularité d'accélérer le développement d'une application grâce notamment à des propriétés de réutilisation. Dans la partie contribution, chapitre 5, nous nous basons sur ce paradigme de programmation pour proposer une solution à notre problématique.

Quelques définitions

La programmation par composant (CBSE [12]) a été introduite dans les années 60 [McI68] et a été réellement exploitée à partir du milieu des années 1990. Cette approche se base sur le principe que tout logiciel peut être représenté par un assemblage de composants. Szyperski [Szy02] donne une des définitions d'un composant la plus citée : "Un composant logiciel est une unité binaire de composition ayant des interfaces spécifiées de façon contractuelle et possédant uniquement des dépendances de contexte explicites". Chaque composant représente donc une entité logicielle indépendante qui a ses propres fonctionnalités, son propre cycle de vie etc. En tant qu'intégrateur, l'implémentation des composants ne nous intéressent pas. En effet, un composant peut être vu comme une boîte noire qui fournit une ou plusieurs fonctionnalités exposées à travers des interfaces, figure 4.3. L'intérêt n'est pas de savoir comment le composant est implémenté mais comment nous pouvons l'intégrer dans une application et le faire interagir avec d'autres composants.

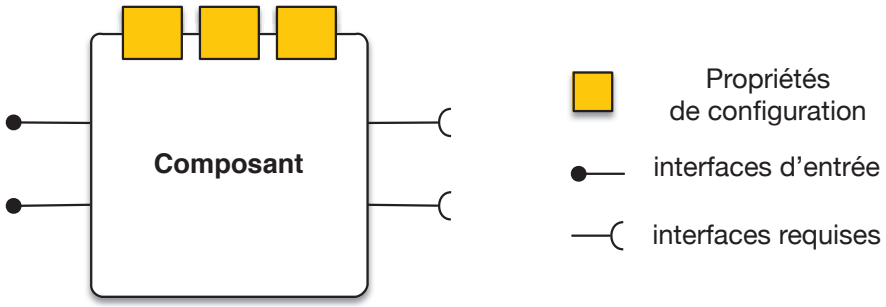


FIGURE 4.3 – Représentation classique d'un composant

La réalisation d'un logiciel basé sur la programmation par composants se fait en deux phases. Dans un premier temps, il faut réaliser les composants qui le structurent. Il faut implémenter la logique des composants et ensuite le décrire à travers ses interfaces. Cette description contient les interfaces d'entrée, les méthodes qui pourront être appelées à partir d'autres composants. Un composant peut aussi dépendre d'autres composants. Ce sont les interfaces qui représentent les méthodes requises pour que le composant fonctionne correctement. Enfin, les composants peuvent être paramétrés à travers diverses propriétés de configuration. Toutes ces informations doivent se trouver dans la description des composants. Dans un second temps, un intégrateur s'occupe de configurer et d'assembler les composants les uns avec les autres. Pour ce faire, il se base uniquement sur les interfaces du composant sans pour autant savoir comment le composant est implémenté, figure 4.4. Ce qu'il faut retenir c'est

12. CBSE pour Component Based Software Engineering

que les acteurs qui implémentent les composants ne sont pas obligatoirement ceux qui les utilisent et intègrent dans la réalisation de logiciels.

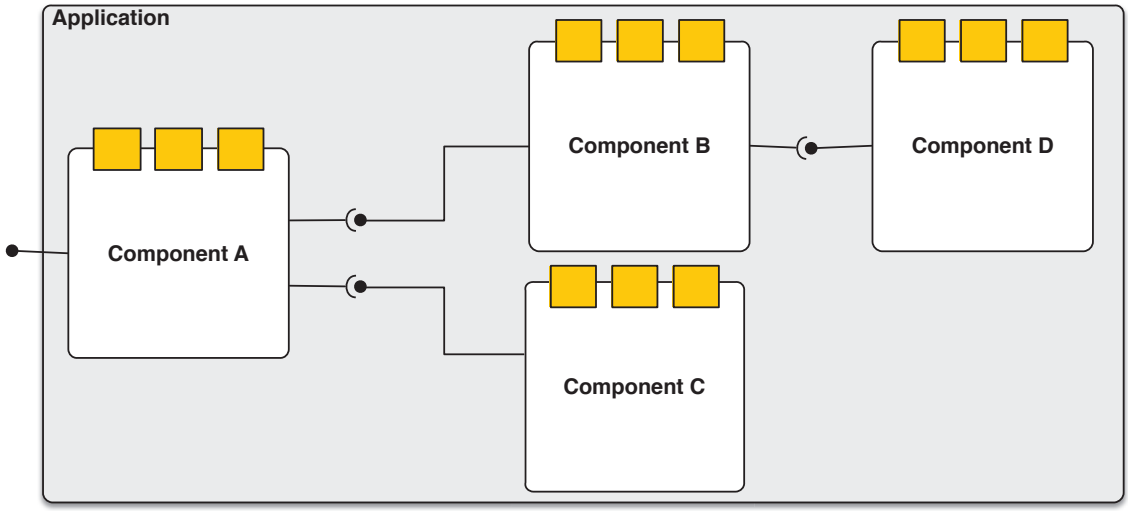


FIGURE 4.4 – Assemblage d'une application à base de composants

Pour assembler les composants entre eux, les intégrateurs se basent sur des modèles à composants. Ces modèles proviennent du monde industriel (**OSGi** [OSG14], **CORBA** [omg06] de OMG [13], **EJB** de SUN [RB10], **COM** de Microsoft [Box97], etc.) et du monde académique (**iPOJO** [EHL07], **Fractal** [BCL+06], **SOFA** [PBJ98], etc.). Dans ces modèles, iPOJO a notamment été porté sur Android. Dans tous ces modèles à composants la description des composants est très importante. C'est grâce à elle que les composants seront reliés.

Pour ce faire, en fonction du modèle, la description des composants se fait par différents moyens mais est toujours basée sur le même principe. La description reprend les fonctionnalités du composant, ses méthodes publiques en POO (programmation orientée object), pour les transposer dans un langage de description. Chaque technologie a son propre langage de description. CORBA utilise, par exemple, les IDL (Interface definition language) [omg14] alors que Fractal utilise un ADL [14] (architecture definition language). Dans le cas de CORBA, seules les interfaces du composant sont représentées alors qu'avec Fractal ADL, en plus de la description des interfaces des composants, l'assemblage des composants est représenté. Nous sommes à un niveau d'abstraction plus élevé. Ainsi, plusieurs langages de descriptions ont été définis (**CORBA IDL**, **Fractal ADL**, **xADL** [DvdHT02], **SOFA 2** [PV02], **C2ADEL** [MRT99] etc.).

Pour faciliter la création de ces interfaces, certaines technologies ont intégré les annotations pour la création de leurs composants. Par exemple, depuis la version 3 des EJBs [EJB08], la description des composants peut se faire directement dans le code source du composant. les développeurs de composants ajoutent devant leurs classes, leurs méthodes etc. des annotations qui représentent les métas données du composant. Dans ce cas, les annotations sont utilisées de façons complémentaires à la technologie de développement utilisée (souvent le Java).

Aujourd'hui, ces différents langages de description ou les technologies complémentaires tels que les annotations peuvent être considérés comme bas niveaux [ZUV10]. Effectivement, ils représentent les méthodes informatiques du composant sans faire abstraction des plates-

13. OMG : Object Management Group
14. Fractal ADL Tutorial : http ://fractal.ow2.org/tutorials/adl/

formes de développement du composant. Dans la plupart des cas, la description ne correspond qu'à une seule implémentation pour une plate-forme spécifique. Il n'est pas possible, par exemple, d'implémenter le même composant en Objective-C et en Java et d'obtenir la même interface de description, figure 4.5. Dans le chapitre 6, nous décrivons un nouveau moyen pour décrire un même composant implémenté dans plusieurs environnements de développement. La description se fait alors à un plus haut niveau d'abstraction qui est indépendant d'une plate-forme de développement particulier.

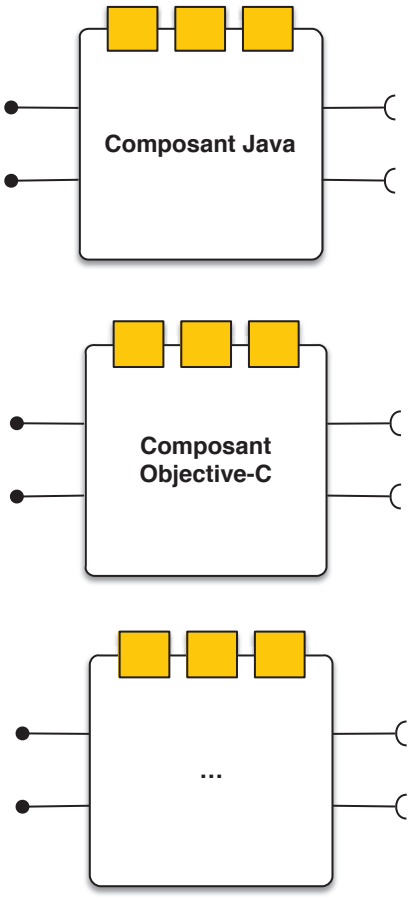


FIGURE 4.5 – Représentation de plusieurs composants implémentés à partir de différents environnements de développement

Gains apportés par la programmation par composants

Les gains de la programmation par composants sont nombreux et permettent d'accélérer le temps de développement d'une application. Ceci s'explique notamment à travers la réutilisation des composants. Un composant fournit, en effet, un service à travers plusieurs fonctionnalités. Ce service peut être alors utilisé et ensuite réutilisé dans plusieurs applications en fonction de leurs besoins. Dans le développement mobile, certains services sont implémentés dans pratiquement chaque application. Par exemple, un service de gestion de la localisation ou un service d'envoi de requêtes HTTP à des web services sont utilisés couramment dans un grand nombre d'applications. Pour autant, il est rare que ces services soient mutualisés entre plusieurs applications. Cela s'explique notamment par le fait que lorsqu'une personne a une idée d'application mobile, le temps entre la formalisation de l'idée et la sortie de l'application mobile qui y répond est très court. Parfois cela est espacé d'à peine trois mois

en fonction de la complexité de l'application. De plus, il est très rare pour une entreprise non spécialisée dans le développement mobile de réaliser plusieurs applications mobiles. Dans ce cas, les entreprises n'arrivent pas à mutualiser leurs développements. Chez Keyneosoft, et plus généralement dans toutes les entreprises de développement mobiles, nous sommes dans le cas contraire. Nous éditons des logiciels mobiles. Par conséquent, la mutualisation de nos développements est capitale. Les composants sont à priori un bon investissement.

En plus de la réutilisation, les applications à base de composants sont plus faciles à maintenir qu'une application classique. Par exemple, si un composant contient un bug, il suffit de le remplacer dans toutes les applications qui l'utilisent, figure 4.6. Cette substitution n'a aucun impact sur l'application. En effet, lors de la correction d'un bug, seule l'implémentation du composant est modifiée. Quant à elles, les interfaces du composant, ne changent pas. Par conséquent, les liens entre les composants de l'application ne changent pas. Pour le développement mobile, cette propriété est très intéressante. Les SDKs mobiles changent en effet souvent (une fois par an minimum) avec des impacts plus ou moins grands. Par exemple, sur Android 2.0, l'appel à un web service pouvait se faire sur le thread principal du système d'exploitation. C'est ce thread qui gère entre autres l'interface utilisateur des applications. Ce genre d'appel avait la particularité de bloquer complètement l'interface utilisateur. Dans les versions supérieures d'Android, ce système a été interdit. Par conséquent, les entreprises de développement qui effectuaient ce genre de processus ont dû pour chacune de leurs applications revoir leur système d'appel au web service. Si, elles avaient à ce moment-là, un composant d'envoi de requêtes HTTP, il aurait suffi de changer son implémentation et de le réimplanter dans toutes leurs applications sans aucun changement. Le temps gagné en maintenance est alors considérable.

La substitution des composants peut aussi se faire à la volée, pendant l'exécution de l'application références [HMT+04]. Ce procédé est utilisé, par exemple, pour créer des applications sensibles au contexte [PDLP11]. Par exemple, si une application n'a plus de réseau, certains composants ayant besoin du réseau peuvent être remplacés par d'autres composants qui fournissent le même service sans réseau. Souvent, les services sans réseau fournissent de moins bons résultats. Cependant, l'application fonctionne toujours même si il n'y a pas de réseau. La substitution se fait de manière transparente pour l'utilisateur.

Enfin, la programmation par composant permet d'augmenter la qualité des logiciels développés. Au bout d'un certain temps, lorsqu'un composant est mature, après quelques années et quelques intégrations dans des applications, le composant peut être considéré comme très fiable. Plus une application contient des composants fiables, plus elle sera elle-même fiable.

Discussions

Malgré toutes ces propriétés intéressantes ce genre d'approche est difficile à mettre en place dans une entreprise de développement mobile. Les applications pour smartphones sont relativement jeunes. Elles sont souvent relativement petites (quelques écrans, deux ou trois fonctionnalités). Elles sont donc implémentées rapidement. Généralement, cela ne dépasse pas les trois mois de développement. La programmation par composant implique que la conception et l'analyse des projets se fassent en profondeur dans le but d'identifier les composants d'une application et surtout d'identifier les composants qui pourront être réutilisés dans d'autres applications. Aujourd'hui, les temps de conception des projets mobiles ne sont pas forcément adaptés à ce type d'analyse.

Cela est en train de changer. Depuis un ou deux ans, la demande d'applications mobiles concerne de plus en plus des applications complexes et plus uniquement des applications mo-

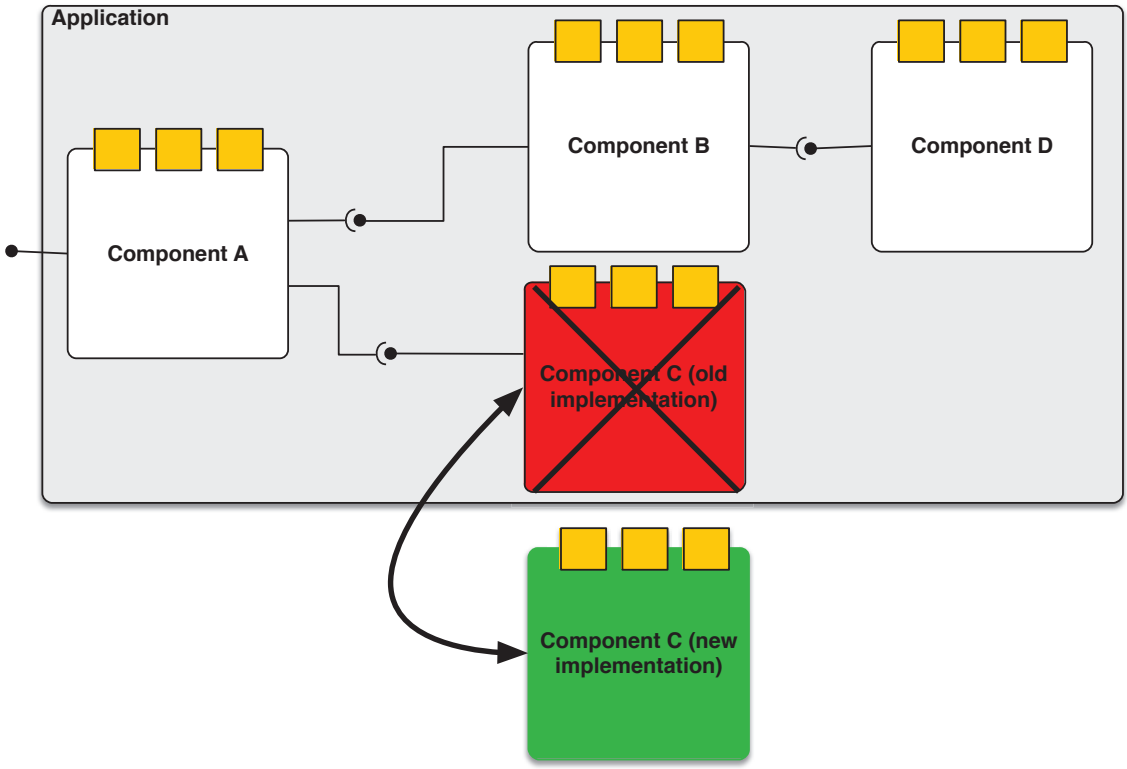


FIGURE 4.6 – Remplacement d'une implémentation d'un composant par une autre

biles qui sont uniquement des gadgets ou des applications pour faire le buzz. Effectivement, les d'entreprises ont compris qu'avec des applications mobiles, elles pourraient accroitre leur productivité. Ce type d'application est souvent exclusivement disponible pour les collaborateurs d'une entreprise. Par exemple, une application mobile peut permettre à un vendeur d'avoir avec lui son catalogue "produits" mis à jour en temps réel. Il pourra alors le présenter à travers un support plus ludique qu'un catalogue papier. Il aura en effet plus d'images que sur le catalogue papier, des vidéos, des graphiques etc. De plus, ce genre d'application peut être agrémenté de fonctionnalités telles que la prise de commande à partir du catalogue, la réservation de produits, etc. Toutes ces fonctionnalités permettent aux vendeurs d'accroître leur productivité. Ce type d'application peut être développé pour beaucoup d'autres domaines. Pour ce genre d'application, le développement est beaucoup plus long. Le temps de conception l'est d'autant plus. Il est donc devenu envisageable d'adopter une approche telle que la programmation par composant.

4.5 Conclusion

Dans ce chapitre, nous avons passé en revue les solutions de plus haut niveau que dans le chapitre précédent. Dans ces solutions, la définition de l'architecture de l'application est au centre des préoccupations. D'un côté, nous avons étudié les langages spécifiques à un domaine et l'ingénierie dirigée par les modèles. Pour ces deux approches, il a fallu étudier le domaine du mobile pour en faire ressortir les caractéristiques et créer soit un langage de programmation unique, soit un langage de modélisation commun à toutes les plates-formes mobiles. Dans les deux cas, aucune étude ne permet de savoir si les développeurs gagnent réellement du temps avec ce genre de solution. De plus, les applications générées peuvent être

plus ou moins performantes en fonction de l'éloignement entre le langage commun et la plate-forme visée. Enfin, ce genre de solutions n'offre pas la flexibilité attendue pour le domaine du mobile qui évolue rapidement. Effectivement, ces solutions dépendent d'un compilateur par plate-forme visée. Le lien entre le langage de haut niveau et chacun des SDKs peut être difficile à mettre en place. Nous avons déjà relevé ce problème avec les compilateurs sources à sources. D'un autre côté, nous avons étudié les solutions qui se basent sur un nouvel environnement d'exécution installé sur les plates-formes mobiles existantes. C'est le cas des machines virtuelles ou du portage d'un système d'exploitation. Cependant, ces solutions ne peuvent pas être exploitées dans un cadre professionnel. Elles dépendent, en effet, du bon vouloir des fournisseurs de systèmes d'exploitation. Une autre approche est de diviser l'exécution d'une application entre deux plates-formes, le système d'exploitation mobile et le cloud. Cela pose un problème pour les utilisateurs. Ils doivent, en effet, toujours être connectés à internet pour que leurs applications fonctionnent.

Pour terminer, nous avons présenté la programmation par composants. Ce style de programmation permet en divisant l'application en plusieurs composants indépendants et réutilisables d'accroître la productivité des développeurs. Il permet aussi d'augmenter la qualité des logiciels produits et par la même occasion de simplifier la maintenance. Cependant, aujourd'hui, aucun modèle à composants n'a été adapté pour le domaine du mobile et plus particulièrement pour le développement mobile d'applications multiplateformes. Dans le prochain chapitre, nous présentons un cadre de développement basé sur cette approche. Ce cadre de développement permet d'intégrer des composants mobiles multiplateformes à partir d'un langage commun dans des applications mobiles implémentées avec différents langages de programmation.

Deuxième partie

Contributions

Chapitre 5

La programmation par composants pour le développement d'applications mobiles

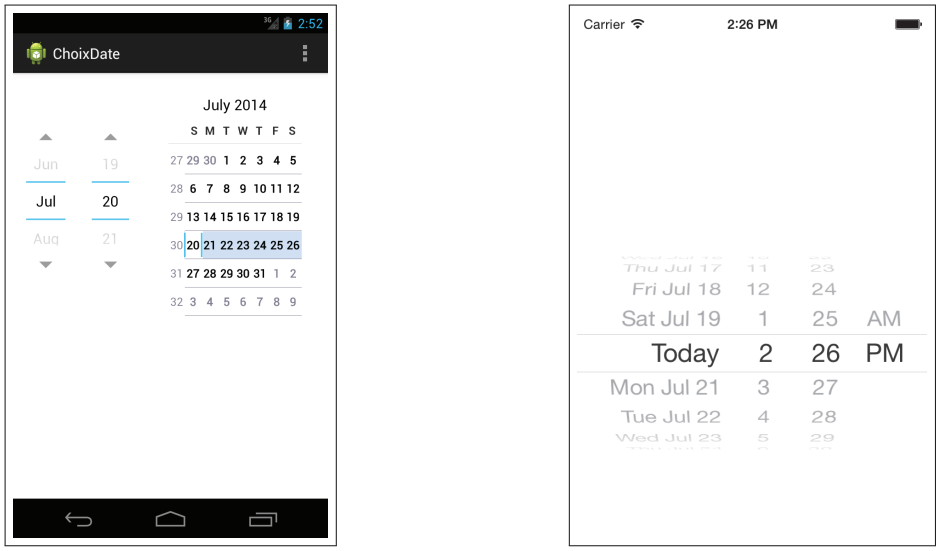
Depuis plusieurs années, la programmation par composants a montré des avantages certains pour le développement ainsi que pour faciliter la maintenance de logiciels en favorisant la réutilisation de composants. Cependant, aujourd'hui, cette approche n'a pas encore été utilisée pour adresser l'hétérogénéité des mobiles. Dans ce chapitre, nous allons donc introduire ce type de programmation pour le développement mobile multiplateforme. Nous verrons que les règles strictes des fournisseurs de systèmes d'exploitation mobiles imposent des contraintes sur le modèle classique de ce type de programmation.

Sommaire

5.1 Introduction 61
5.2 Architecture générale 63
5.2.1 Couche 1 : Structure de l'application 65
5.2.2 Couche 2 : langage universel 66
5.2.3 Couche 3 : Un panel de composants multiplateformes 68
5.3 Compilation et applications générées 70
5.4 Réutilisation 71
5.5 Conclusion 72

5.1 Introduction

Comme nous l'avons vu dans le précédent chapitre, la programmation par composants permet de diminuer le temps de développement en favorisant leur réutilisation. Elle permet aussi de maintenir plus efficacement une application. Aujourd'hui, ces deux points sont nos principaux objectifs. Nous avons donc décider de suivre ce modèle de programmation et de l'utiliser dans le cadre du développement mobile multiplateforme. Pour ce faire, nous allons



(a) Affichage d'un calendrier sur Android (b) Affichage d'un calendrier sur iOS

FIGURE 5.1 – Éléments natifs pour le choix d'une date sur iOS et Android

proposer aux développeurs utilisant notre solution un moyen universel d'intégrer un panel de composants dit multiplateformes dans n'importe quelle application mobile.

Dans le chapitre 2, section 2.1, nous avons présenté les contraintes liées au développement mobile multiplateforme. Pour rappel, il faut que la solution proposée fournisse les mêmes propriétés que celles fournies par le développement natif (accès aux capteurs mobiles, accès aux notifications pushs, cloud, etc.) à la différence des solutions existantes vu dans les chapitres 3 et 4. En effet, les applications doivent proposer une expérience utilisateur aussi riche que celle native. Pour cela, il faut que les applications générées soient performantes et réactives. De plus, elles doivent suivre les règles de chaque plate-forme en terme d'interface et d'ergonomie.

Au niveau du développement, une des règles la plus contraignante est l'obligation d'utiliser le langage de programmation natif pour avoir la possibilité d'être publié sur les magasins d'applications officiels de chaque plate-forme cible. Par exemple, nous ne pourrons pas soumettre une application sur l'App Store d'Apple si elle est programmée en Java. De plus, la solution doit être flexible. Elle doit s'adapter aux évolutions de chaque plate-forme rapidement après leurs sorties. Enfin, les outils de développements doivent être aussi complet que ceux natifs (debugger, break point, etc.)

En plus de respecter ces contraintes, nous voulons offrir plusieurs fonctionnalités aux développeurs utilisant notre solution.

— Elle doit fournir un langage commun pour intégrer nos composants dans une application native. Dans le chapitre précédent, section 4.4, nous avons montré que les modèles à composants utilisent un seul langage pour assembler les composants entre eux.
— Nos composants doivent s'intégrer parfaitement dans l'environnement d'exécution de chaque plate-forme cible. Ainsi, si nous sommes sur Android, nos composants (graphiques ou logiciels) doivent respecter l'aspect et le fonctionnement requis sur la plate-forme Android. Par conséquent, pour la même fonctionnalité d'un composant, il sera possible en fonction de la plate-forme de ne pas passer par les mêmes processus internes. Par exemple, si nous exposons un composant graphique dans lequel il est possible de choisir une date, figure 5.1, l'aspect graphique sera différent sur chaque plate-forme cible même si le résultat sera le même.

Dans ce chapitre, nous allons donc présenter un nouveau framework de développement basé sur la programmation par composant dans le but de faciliter le développement multiplateforme d'applications mobiles. Dans un premier temps, nous allons présenter son architecture générale pour ensuite décrire chacun des éléments de notre système. Nous finirons enfin par une présentation de la structure des applications générées avec ce framework et une conclusion.

5.2 Architecture générale

Nous illustrons l'architecture que nous souhaitons mettre en place sur la figure 5.2. Cette architecture a pour objectif de faciliter le développement multiplateforme et d'atténuer les différences entre chaque plate-forme cible. Avec notre solution, nous permettons aux développeurs d'intégrer un nouveau type de composants, des composants multiplateformes, dans n'importe quelle application (Android, iOS, Windows Phone 8, etc.) [PDL13, oPDL14]. Pour faciliter et diminuer le temps de développement, nous leur permettons de les intégrer à partir d'un langage universel (commun à chaque plate-forme cible). Grâce à ce langage, les développeurs peuvent mutualiser une partie de leurs développements. De plus, les composants leurs seront fournis. Cela signifie que plus ils intègrent de composants, plus leurs charges de développement diminuent.

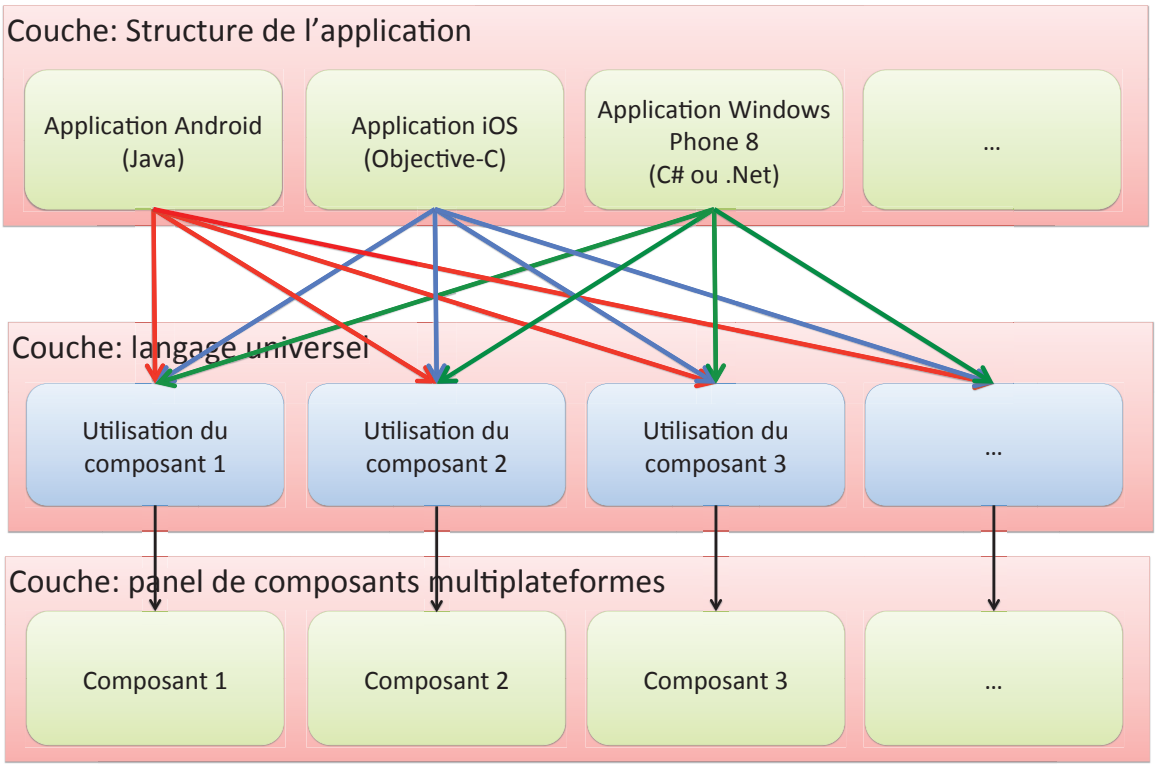


FIGURE 5.2 – Architecture générale de notre framework de développement permettant la prise en compte de l'hétérogénéité des systèmes d'exploitation mobiles

Notre solution est divisée en trois couches distinctes :

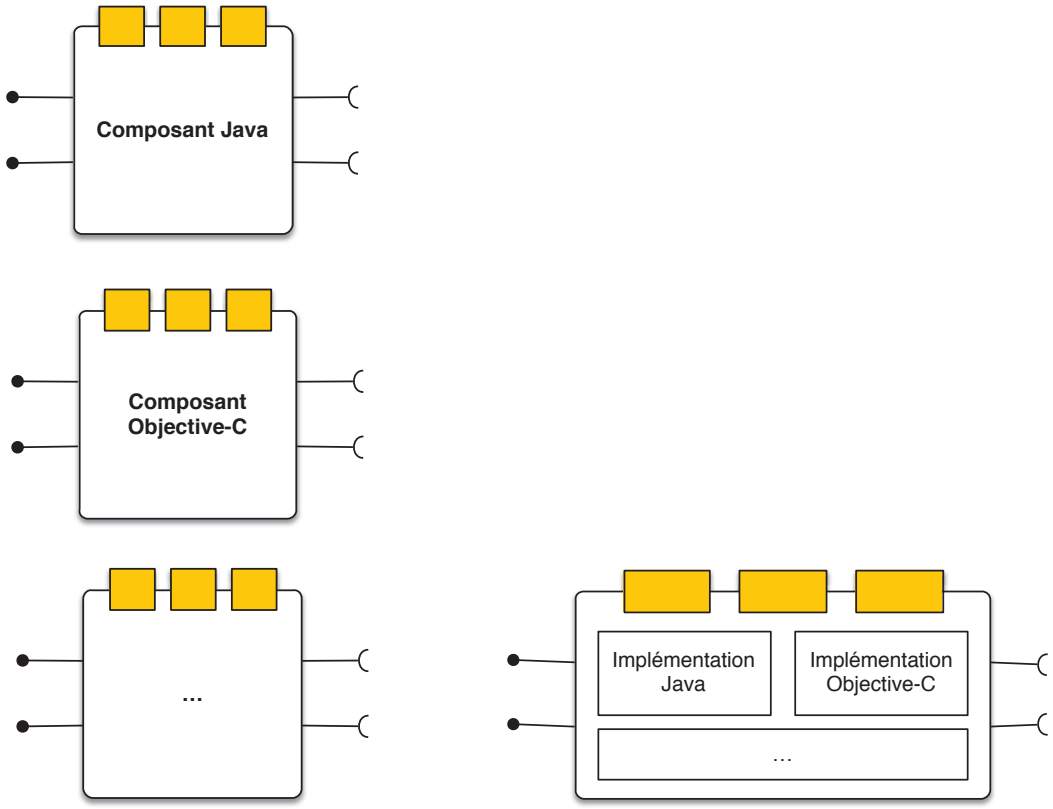
— La première couche correspond à la **structure de l'application**. La structure de l'application est implémentée autant de fois qu'il y a de plates-formes cibles. Elle est implémentée par les développeurs de façon native, c'est-à-dire, avec les outils officiels de chacune des plates-formes cibles. Sur Android, elle sera implémentée en Java avec

Eclipse. Sur iOS, elle sera implémentée en Objective-C ou en Swift avec XCode. Sur Windows Phone, elle sera implémentée en C# ou .Net avec Visual Studio.

— La deuxième couche se compose d'un **langage universel** permettant d'intégrer et d'appeler les fonctionnalités de nos composants de la même façon sur chacune des plates-formes cibles. La même invocation pourra être réutilisée dans n'importe quel code source natif pour mobile.

— La troisième couche se compose d'un **panel de composants multiplateformes**. Ces composants sont intégrables sur n'importe quelle plate-forme mobile et peuvent être intégrés de façon unique avec notre langage universel. En plus de cela, ces composants sont réutilisables dans n'importe qu'elle application mobile.

Pour que les composants soient intégrables dans n'importe quel environnement d'exécution, ils ont la particularité d'avoir une implémentation par plate-forme cible avec une seule interface publique. En reprenant la représentation classique d'un composant, nous obtenons une nouvelle représentation adaptée à notre solution et à la problématique d'hétérogénéité, figure 5.3. Dans les modèles à composants actuels, la représentation du composant est très proche de son implémentation, figure 5.3a. Dans notre solution, figure 5.3b, nous faisons abstraction de la plate-forme de développement du composant. Le fait fait d'avoir une seule interface permet d'y accéder à partir de n'importe quel environnement de développement (Android, iOS, Windows Phone, etc.).



(a) Composant spécifique à une plate-forme (b) Composant multiplateforme

FIGURE 5.3 – Différences entre un composant classique et un composant multiplateforme

Enfin, les applications générées avec notre solution sont 100% natives. C'est-à-dire que leurs exécutables ne contiennent que du code natif. Pour cela, nous allons fournir un compilateur qui s'occupera de transformer le code écrit avec notre langage en langage de program-

mation de la plate-forme cible.

Dans la suite, nous allons présenter brièvement chacune des couches de notre architecture. Elles seront ensuite étudiées plus précisément dans les chapitres 6 et 7.

5.2.1 Couche 1 : Structure de l'application

Avec notre solution, les développeurs doivent implémenter la structure de leur application de façon native sur chacune de leur plate-forme cible. Plusieurs technologies comme PhoneGap [War14], par exemple, fonctionnent sur le même principe. En laissant cette liberté aux développeurs nous avons plusieurs objectifs :

- Permettre l'utilisation des éléments natifs pour l'ergonomie de l'application.
- Permettre l'utilisation des outils officiels pour le débogage.
- Laisser la possibilité d'intégrer leur expertise dans le domaine du mobile qu'ils ont acquise depuis plusieurs années.
- Intégrer des composants, soit dans de nouvelles applications, soit dans des applications existantes. En passant par notre solution, les développeurs pourront intégrer un composant dans une application qui a été développée plusieurs années auparavant.
- Implémenter tout ce qui n'est pas encore disponible sous forme de composants. Le développement de composants a en effet un coût. Il est donc facile de comprendre que tous les composants pouvant former une application ne soient pas implémentés au moment où nous en avons besoin. Dans notre solution, dans ce cas, les développeurs pourront pallier à ce manque en développant eux même le code manquant.
- Permettre la publication des applications générées sur les magasins d'applications officiels de chaque plate-forme cible.

Tout d'abord, en laissant la possibilité aux développeurs d'implémenter la structure de leurs applications de façon native, nous leurs permettons d'intégrer tous les éléments natifs fournis par chacune des plates-formes cibles pour le développement de l'interface graphique et de l'ergonomie de leurs applications. Nous permettons ainsi aux développeurs de créer une application qui s'intégrera parfaitement sur chaque plate-forme cible au niveau interface utilisateur. Les applications garderont donc l'identité de la plate-forme cible. Pour rappel, dans le chapitre 2, section 2.5, nous illustrons les différences de navigation possibles entre iOS et Windows Phone 8. De plus, la génération d'interface graphique et de l'ergonomie de façon multiplateforme pour applications mobiles sont des travaux à parts entières qui doivent être traités pendant la durée d'une thèse [CCT+03, Van05]. Dans nos travaux, nous déléguons donc le développement de cette partie aux développeurs. Par la suite, nous pourrons aussi créer des composants graphiques. Cependant, ce n'était pas dans les objectifs de cette thèse . D'après notre expérience, chaque application a un design particulier. Il sera donc difficile, à partir d'un composant, de répondre à tous les cas. Dans les perspectives de cette thèse, nous verrons qu'il pourrait être intéressant de combiner notre approche avec des travaux sur l'IHM.

Ensuite, nous voulons laisser la possibilité aux développeurs d'utiliser les technologies officielles pour déboguer leurs applications. En effet, les applications générées avec notre solution sont 100% native, c'est-à-dire, que le code contenu dans l'exécutable de l'application sera uniquement composé de langage natif : Java pour Android, Objective-C ou Swift pour iOS et C# ou .Net pour Windows Phone 8. Ainsi, les outils de débogage seront complètement utilisables et pourront facilement interpréter les applications générées. Aujourd'hui, plusieurs technologies existantes qui fournissent des applications hybrides ne permettent pas l'utilisation des outils officiels de débogage. Par exemple Phonegap est dans ce cas. Le code

écrit avec cette technologie (HTML, CSS, Javascript) ne peut pas être débogué avec les outils officiels. Cela engendre de vrais manques dans la phase de développement et de maintenance des applications. Nous n'avons pas ce problème avec notre solution.

De plus, l'intégration de nos composants se fait à la demande dans le code de l'application. Notre solution se veut très flexible et complémentaire à la programmation native. Si un développeur ne veut intégrer qu'un seul de nos composants et écrire tout le reste de son application en langage natif, il pourra le faire. Cela permet de n'avoir aucune limite. En effet, si une fonctionnalité est requise pour une application mais qu'aucun de nos composants ne puisse y répondre, il faut que le développeur puisse avoir une alternative à cela sans pour autant abandonner notre solution. Dans le même ordre d'idée, dans les entreprises d'applications mobiles, il y a souvent des applications qui ont été réalisées il y a deux ou trois ans et qui évoluent encore. C'est par exemple le cas chez Keyneosoft. Avec notre solution, il est tout à fait possible d'intégrer nos composants avec notre langage commun sans pour autant obliger l'entreprise à recommencer l'application de zéro. Enfin, il n'est pas rare que les entreprises dans le domaine du mobile soient spécialisées dans un domaine particulier et aient leurs propres logiciels développés nativement pour ce domaine. Dans le cas de Keyneosoft, nous sommes leaders dans les applications de self-scanning. Pour cela, nous avons implémenté un logiciel permettant de scanner plusieurs types codes barres (EAN-13, EAN-8, QR-Code, etc.) pour Android, iOS et Windows Phone 8. Dans le cas d'autres entreprises, il pourrait s'agir de logiciels de lecture de PDF pour les entreprises spécialisées dans la lecture de documents, ou encore de logiciels de lecture de vidéos pour les entreprises spécialisées le streaming. Il nous fallait avoir la possibilité d'utiliser les logiciels qui représentent l'identité d'une entreprise avec notre solution de développement multiplateforme. La complémentarité de notre framework avec la programmation native est un point très important pour l'adhésion de notre solution dans le monde industriel.

Pour finir, en utilisant les outils officiels de chaque plate-forme pour la création et le développement de la structure des applications utilisant notre solution, nous nous assurons que les applications générées seront publiables sur les magasins d'applications officiels de chaque plate-forme. En effet, pour déployer une application sur l'App Store, il faut que celle-ci soit générée à partir d'XCode. Cette contrainte est aussi présente pour la publication d'applications Windows Phone. Sur Android, Google laisse le choix de l'environnement de développement. Cependant, il n'est pas réellement possible de programmer pour Android sur XCode ou Visual Studio.

5.2.2 Couche 2 : langage universel

La deuxième couche de notre architecture se base sur un langage de programmation universel. En effet, tout le code écrit avec ce langage peut être réutilisé et intégré dans plusieurs codes source natifs. Comme nous le montrons sur la figure 5.2, dans notre architecture, notre langage a pour objectif de faire le lien entre une application native et notre panel de composants. En effet, il permet d'intégrer dans n'importe quelle application native nos composants multiplateformes avec le même code source que ce soit sur Android, iOS, Windows Phone 8, etc. Plus exactement, ce langage va permettre d'accéder aux fonctionnalités de chacun des composants. Dans les prochains chapitres, nous verrons comment ces interactions sont possibles.

Notre langage a trois particularités permettant de faciliter le développement multiplateforme :

— Les invocations effectués avec ce langage sont communs à n'importe quelle plate-forme.
— Il est complémentaire au langage natif.

— Il est facilement transformable en langage natif.

Tout d'abord, notre langage permet d'appeler n'importe quelle fonctionnalité d'un composant à partir d'un seul code source. Ce code source est alors intégrable dans n'importe quel code source natif. Aujourd'hui, les solutions existantes ne permettent pas d'invoquer un composant de la même façon à partir de plusieurs codes sources écrits dans des langages différents. Prenons l'exemple d'un appel à un même web service dans une application Java et C#. Lorsque nous voulons l'appeler en Java, il faut un client Java qui s'occupera de gérer l'appel. De la même façon, il y aura un autre client C# pour gérer l'appel dans l'application C#. Dans, ces conditions les développeurs qui appellent le même web service à partir d'une application Java et d'une autre C# utilisent un client différent, figure 5.4. Ils doivent donc connaitre les spécificités de chacun des clients dans deux langages différents. Dans notre problématique, il est important pour nous de changer cette façon de faire. Comme nous l'avons souligné, dans le chapitre 2, section 2.2, les entreprises de développement mobiles doivent former leurs développeurs à plusieurs technologies différentes. Avec notre langage commun, un développeur Android ayant des compétences dans notre langage, pourra maintenir les invocations à nos composants dans une application iOS, Windows Phone 8 ...il n'aura pas obligatoirement besoin de connaitre les spécificités de chacune des plates-formes cibles pour le faire. Ça n'aurait pas été le cas, si nous fournissons pour chaque langage un client différent pour appeler chacun de nos composants. Avec ceci, nous diminuons le temps de développement et facilitons la maintenance des applications utilisant notre solution.

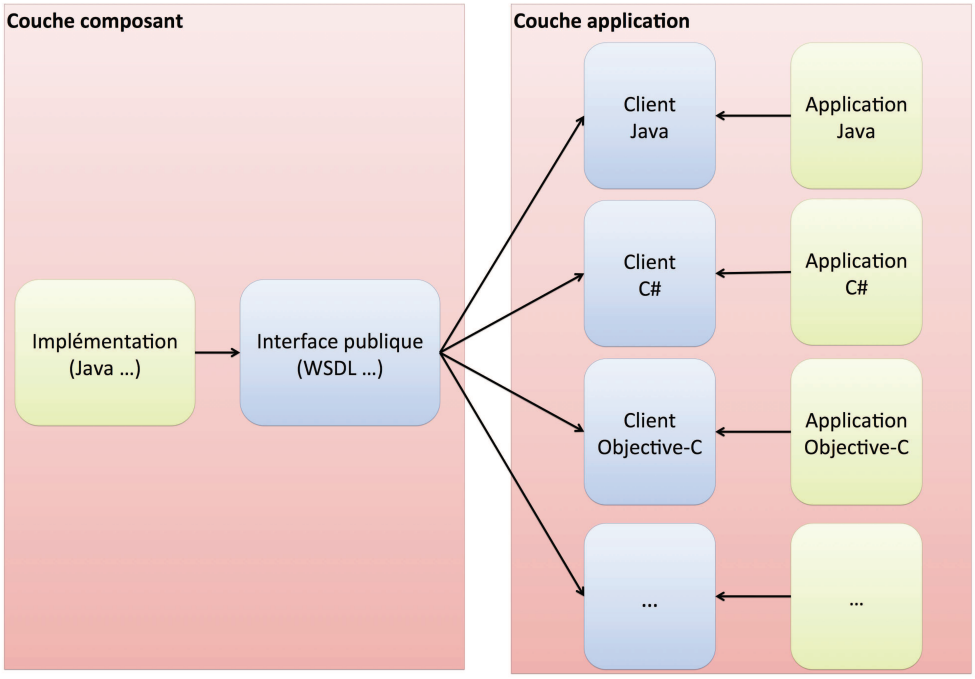


FIGURE 5.4 – Utilisation de composants classiques à partir d'applications de bureau

Dans la section précédente, nous avons insisté sur les avantages de permettre l'implémentation de la structure d'applications en langage natif. Notre langage a donc la particularité d'être complémentaire au langage natif. Il ne permettra donc pas d'implémenter une application entière. Cependant, notre langage pourra être intégré n'importe où dans un code source natif. Dans le but de faciliter son intégration dans un code source, nous nous sommes basés sur les annotations. Aujourd'hui, les annotations sont très utilisées que ce soit sur Android, iOS ou Windows Phone. En choisissant ce type de langage, nous pensons que l'adhésion à notre langage sera plus forte qu'en créant un nouveau langage. De plus, il n'y aura pas de

surcoût élevé à son apprentissage. Dans le prochain chapitre, nous revenons plus précisément sur les possibilités et propriétés qu'offrent notre langage.

Avec notre architecture, nous voulons générer uniquement des applications 100% natives. Cela implique que notre langage commun doit être transformé à la compilation en code natif. Pour cela, il nous faudra un compilateur qui pourra, dans n'importe quel code source, transformer le code écrit avec notre langage en code natif. Nous verrons dans les prochains chapitres que nous nous sommes focalisés sur la flexibilité de ce compilateur dans le but de prendre en charge le plus rapidement possible les évolutions rapides et inattendues du domaine mobile (nouvelle sortie d'une version d'un système d'exploitation, nouvelle sortie d'un langage de programmation, nouvelle sortie d'un système d'exploitation).

5.2.3 Couche 3 : Un panel de composants multiplateformes

La dernière couche de notre architecture correspond à un panel de composants multiplateformes que nous fournissons aux utilisateurs de notre solution. Classiquement, dans la programmation par composants, le point d'entrée d'un composant est représenté par une interface publique qui fait le lien avec son implémentation. Sur la figure 5.4, nous montrons un exemple d'utilisation d'un composant à partir de plusieurs logiciels développés dans des langages différents. Pour utiliser le composant, il faut pour chaque environnement de développement cible créer un client logiciel permettant d'accéder au composant. Avec les contraintes liées aux fournisseurs de systèmes d'exploitation mobiles et à leur environnement d'exécution, nous ne pouvons pas nous contenter de ce système. En effet, sur un ordinateur de bureau avec Windows, Linux ou Mac OS X installé, il est tout à fait possible de lancer un composant Java à partir d'une application C#. Les deux environnements d'exécution peuvent cohabiter sur ces systèmes d'exploitation. Cette configuration est impossible sur mobile. Nous ne pourrons, par exemple, pas exécuter un composant Android (développé en Java) sur iOS. Il n'existe pas d'environnement d'exécution Java pour iOS. Dans notre solution, un composant ne pourra donc pas avoir une unique implémentation.

Sur la figure 5.5, nous montrons quelles sont les modifications à apporter pour utiliser nos composants dans notre framework de développement. Pour respecter les contraintes d'exécution fournies par les systèmes d'exploitation mobiles, nos composants ont plusieurs implémentations : une par plate-forme cible. Par rapport à la programmation par composants classiques, nous ajoutons la notion de client unique grâce à notre langage universel. Nous gardons bien-sûr l'interface publique unique pour chacun de nos composants. Sans ça, nous ne pourrions pas faire un client unique. Cette interface publique présente les fonctionnalités de nos composants de façon indépendante d'une plate-forme cible. Le composant pourra donc être appelé à partir de plusieurs environnements de développement.

Cette nouvelle représentation d'un composant révèle plusieurs problématiques. En effet, aujourd'hui, dans la programmation par composant classique, il est facile de représenter les fonctionnalités d'un composant dans une interface publique. Il suffit de reprendre les méthodes que l'on veut exposer et les transformer dans un format abstrait mais dépendant de la plateforme (IDL [omg14] ou WSDL [CCMSW01], par exemple), voir chapitre 4, section 4.4. Dans notre proposition, nous perdons cette facilité. En effet, nous devons réunir les méthodes provenant de plusieurs SDK et langages de programmation différents pour les exposer de façon unique dans une seule interface. De plus, cela ne fonctionne que lorsque les fonctionnalités ont des équivalences sur chacune des plates-formes cibles. Il est, par exemple, possible que certaines fonctionnalités ne soient pas disponibles sur tous les systèmes d'exploitation. Nous pensons naturellement au NFC qui est disponible sur Android et Windows Phone 8 alors qu'il ne l'est pas sur iOS. Si nous voulons fournir un composant de gestion de NFC, il ne sera

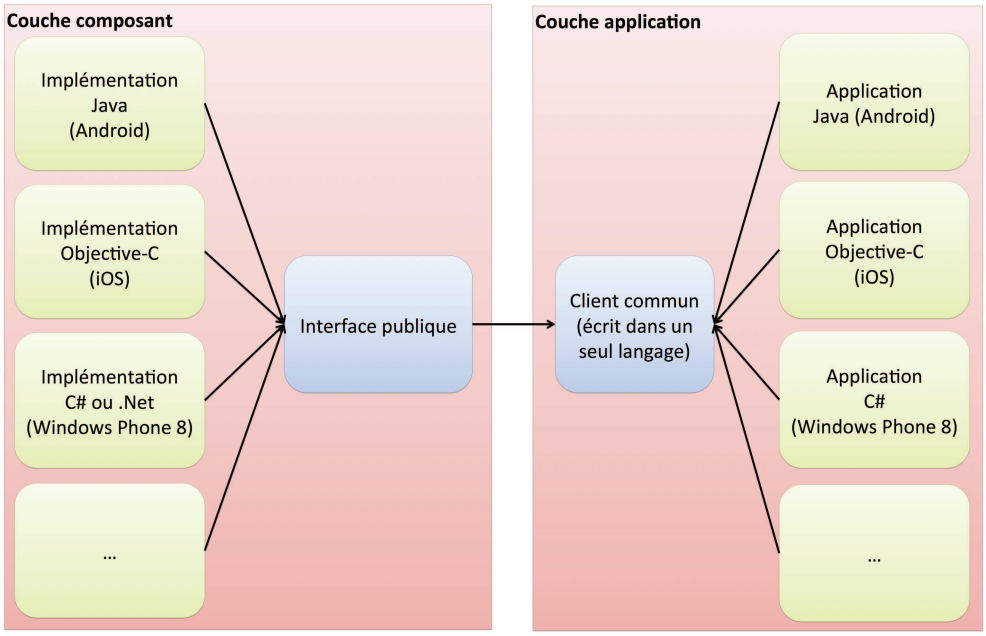


FIGURE 5.5 – Utilisation de composants multiplateformes dans notre solution

disponible que pour Android et Windows Phone 8, il faudra donc trouver une solution pour inclure ou exclure certains systèmes d'exploitation dans l'interface publique du composant. Dans notre proposition, nous permettrons donc la division des fonctionnalités d'un composant en fonction d'une plate-forme particulière.

Pour résumé, nos composants multiplateformes ont :

— Plusieurs implémentations : une implémentation par plate-forme cible. Cela nous permet de respecter les règles d'exécutions fournies par chaque plate-forme cible. Nous verrons dans le prochain chapitre que ce choix nous permet aussi de n'avoir aucune limitation dans l'utilisation des composants présents nativement que ce soit des composants graphiques ou logiciels.

— Une seule interface publique : expose de façon unique les fonctionnalités du composants implémentés de plusieurs façons.

— Un seul client permettant d'accéder à leurs fonctionnalités grâce à notre langage universel. Ainsi, nous diminuons le temps développement et facilitons la maintenance des applications mobiles.

Sur la figure 5.6, nous intégrons les modifications que nous avons apportées à nos composants multiplateformes dans l'architecture de notre solution. Sur cette figure, nous montrons aussi comment notre framework de développement fait le lien entre le code de l'application et une implémentation spécifique d'un composant. Nous remarquons par exemple, que l'application Android et iOS utilise le même code pour appeler nos composants. Pour autant, notre framework différencie chacune des implémentations de nos composants (flèches rouges pour Android et flèches bleues pour iOS). Dans la suite, nous allons présenter la composition interne des applications générées à partir de notre framework de développement.

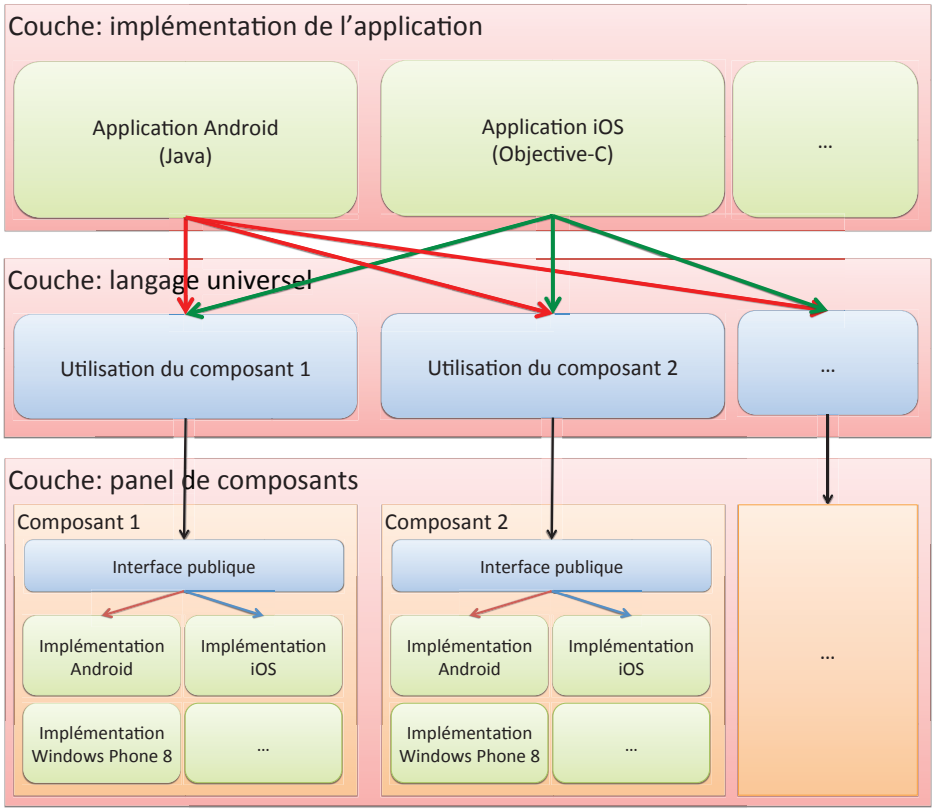


FIGURE 5.6 – Architecture détaillée de notre framework de développement

5.3 Compilation et applications générées

Nous souhaitons que les applications générées avec notre solution ne contiennent que du code source natif. Ainsi, les applications générées seront les plus performantes possibles. Nous n'ajouterons pas de surcoût à l'exécution telle que nous avons pu l'observer dans les solutions existantes. Par exemple, les solutions basées sur les technologies web interprètent le code source HTML, CSS et javascript de l'application à l'exécution ce qui augmente inévitablement les temps d'exécution. De plus, nous respecterons les normes fixées par les stores officiels de chaque plate-forme cible.

Sur la figure 5.7, nous reprenons les exemples d'applications fournis dans la figure 5.6 et montrons le résultat en terme de composition interne lorsque l'application est générée par notre framework de développement. Sur cette figure, nous pouvons voir que la structure de l'application (vues, traitements métiers) est implémentée nativement par l'utilisateur de notre framework, c'est la couche "structure de l'application" de la figure 5.6. Ensuite, les invocations de nos composants avec notre langage commun sont aussi transformées en code natif, c'est la couche "langage universel" de la figure 5.6. Enfin, l'implémentation du composant qui est embarquée avec l'application est celle correspondante à la plate-forme ciblée, c'est la couche "panel de composants" sur la figure 5.6. Nous n'embarquons donc pas tout le composant mais uniquement la partie qui sera utilisée.

Pour réaliser l'exécutable sur chaque plate-forme des applications utilisant nos composants, nous introduisons une notion de pré-compilation. Avant chaque déploiement avec les outils officiels, nous lançons notre compilateur qui va s'occuper d'intégrer dans les applications la bonne implémentation du composant et surtout va transformer le code écrit avec notre langage en code natif. Ensuite, après la transformation des applications avec notre ou-

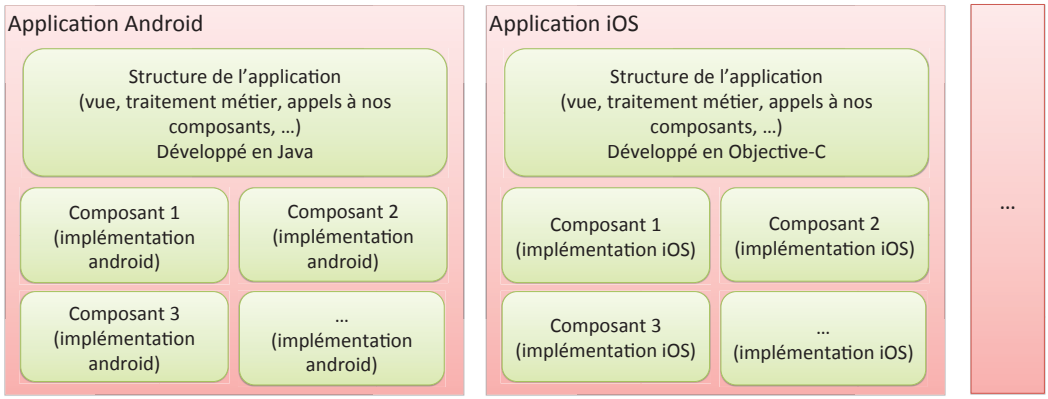


FIGURE 5.7 – Application générée par notre compilateur

til, nous laissons la formation de l'exécutable final aux compilateurs de chaque plate-forme cible.

5.4 Réutilisation

Le gain de cette solution est très important si l'entreprise qui intègre les composants n'est pas celle qui les réalise. Le gain est tout autre si l'entreprise développe et intègre elle-même ses composants. C'est le cas notamment de Keyneosoft. Nous verrons dans le chapitre 9, que la réalisation de composants multiplateformes a un coût non négligeable. Il faut souvent réutiliser 2 ou 3 fois le même composant avant de le rentabiliser. Il faut par conséquent les intégrer dans le plus grand nombre d'applications possibles, figure 5.8.

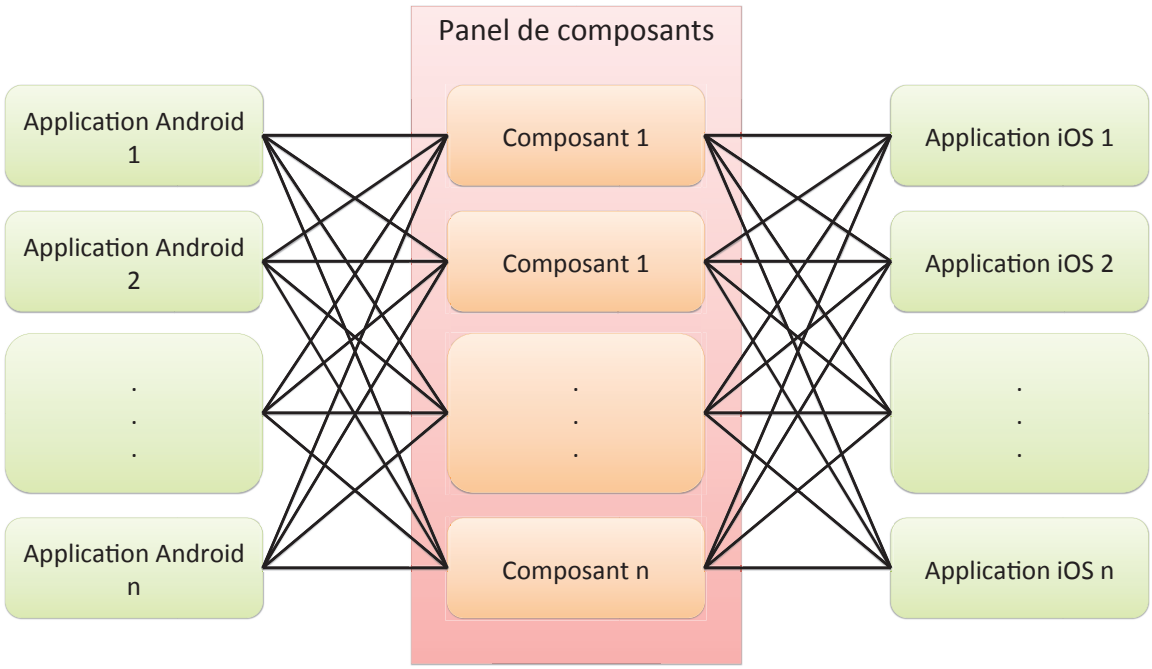


FIGURE 5.8 – Réutilisation des composants dans un maximum d'applications

Le choix et la conception des composants est donc primordial. Il faut que les composants soient indépendants d'une application. Par exemple, un service de localisation ou d'envoi de

requêtes HTTP à un web services sont des services indépendants d'une application. Ils sont donc susceptibles d'être intégrés partout. Si ils dépendent d'une application, il ne seront utilisés qu'une seule fois. Dans ce cas, avec notre solution, il est plus judicieux de les implémenter nativement avec la structure de l'application.

5.5 Conclusion

Dans ce chapitre, nous avons présenté l'architecture du framework de développement que nous souhaitons mettre en oeuvre pour diminuer le temps de développement d'applications mobiles multiplateformes. Pour obtenir ce résultat, nous intégrons la programmation par composant au coeur du développement mobile. Cependant, en l'état actuel, il n'est pas possible d'utiliser simplement les outils existants. Nous devons adapter ce type de programmation aux contraintes fournies par notre problématique et par les fournisseurs de systèmes d'exploitation mobiles.

Nous avons donc adapté l'accès aux composants avec un langage de programmation universel permettant l'appel aux fonctionnalités de chacun des composants. Ce langage commun a la particularité d'être intégrable dans n'importe quel environnement de développement natif et d'être facilement transformable en langage natif.

Grâce à ces deux propriétés, nous permettons aux utilisateurs de notre solution d'implémenter la structure de leurs applications en langage natif. Ainsi, ils pourront utiliser les environnements de développement natifs pour implémenter les vues et l'ergonomie de leurs applications de façon natives. Leurs applications s'intégreront parfaitement dans l'environnement ciblé. De plus, ils pourront maintenir leurs applications à partir des outils de débogage natif, ce qui était un vrai manque dans certaines des technologies étudiées dans le chapitre 3. La maintenance sera plus facile.

Enfin, pour respecter les contraintes des environnements d'exécution mobiles, nous avons modifié la structure des composants classiques. Dans notre solution, un composant a la particularité d'avoir plusieurs implémentations, une par plate-forme cible avec une seule interface publique. Cette nouvelle structure entraine de nouvelles problématiques telle que : "comment réunir en une seule interface plusieurs implémentations ?". Dans le prochain chapitre, nous nous attarderons sur cette question avec une présentation complète d'un nouveau genre de composants : les composants multiplateformes.

Chapitre 6

Définition d'un nouveau genre de composants : des composants multiplateformes

Dans notre framework de développement, un composant multiplateforme a la particularité d'avoir plusieurs implémentations natives. Ainsi, chaque implémentation peut être exécutée sur des environnements d'exécution distincts. Cependant, cette façon de faire pose une nouvelle problématique : comment allons nous unifier l'accès à un composant alors qu'il est développé de plusieurs manières différentes ? Pour s'abstraire des spécificités de chaque plate-forme nous unifions les différentes implémentations de nos composants à travers des interfaces dites publiques et indépendantes de toutes les plate-formes cibles.

Sommaire

6.1 Introduction 74
6.2 Interactions possibles avec un composant multiplateforme 74
6.2.1 Entrée/sortie d'un composant 74
6.2.2 La délégation : un moyen de rendre un composant indépendant de toutes les applications 75
6.2.3 Un exemple concret : "httpRequestManager" 77
6.3 Structure interne d'un composant multiplateforme 79
6.4 Description de la partie cachée d'un composant multiplateforme 80
6.4.1 Une implémentation par plate-forme cible 80
6.4.2 Une interface complémentaire 81
6.4.3 La partie cachée du composant "HttpRequestManager" 81
6.5 Description visible d'un composant multiplateforme 86
6.5.1 Une interface publique 87
6.5.2 La partie visible du composant "HttpRequestManager" 88
6.6 Conclusion 90

6.1 Introduction

Dans le précédent chapitre, nous avons présenté un framework de développement basé sur la programmation par composant pour faciliter le développement multiplateforme. Nous avons identifié trois couches : la structure de l'application, un langage universel et un panel de composants. Dans ce chapitre, nous allons présenter plus précisément la partie panel de composants.

Nous allons tout d'abord présenter les interactions possibles entre les intégrateurs et les composants, section 6.2. Ensuite, nous étudierons la structure interne d'un composant : la partie cachée avec une implémentation native par plate-forme et la partie visible avec une interface publique, sections 6.3, 6.4, 6.5. Pour illustrer nos propos, nous allons prendre un exemple concret de composant. Ce composant se nomme "HttpRequestManager". C'est un des composants les plus utilisés dans nos applications à Keyneosoft. De plus, il peut représenter tous les types de composants possibles.

L'objectif de ce chapitre est de montrer comment nous arrivons à nous abstraire de toutes les plates-formes cibles pour permettre une utilisation unifiée de nos composants sur toutes les plates-formes mobiles existantes.

6.2 Interactions possibles avec un composant multiplateforme

Dans cette section, nous présentons les interactions possibles entre l'utilisateur de notre solution (l'intégrateur de nos composants) et les composants. Pour cela, nous définissons les entrées /sorties possibles de nos composants. De plus, nous allons montrer comment nous concevons nos composants de façon multiplateforme avec un exemple concret. Pour rappel, la conception du composant doit faire abstraction des plates-formes cibles, chapitre 5, section 5.2.3. En effet, nous devons fournir les mêmes fonctionnalités pour plusieurs plates-formes à travers une seule interface commune.

6.2.1 Entrée/sortie d'un composant

Sur la figure 6.1, nous présentons les interactions possibles entre les intégrateurs et nos composants. Ces interactions sont divisées en trois catégories :

- **Méthodes de configuration :** ce sont les méthodes qui permettent à l'intégrateur de paramétrer le composant. Il pourra par exemple paramétrer les couleurs d'un composant graphique à partir de ces méthodes. Ces méthodes sont optionnelles, il se peut qu'il ne soit pas possible de paramétrer un composant.
- **Méthodes d'entrée :** ce sont les méthodes qui permettent à l'intégrateur de lancer les fonctionnalités du composant.
- **Méthodes de sortie :** ce sont les méthodes qui permettent à l'intégrateur de recevoir les résultats des fonctionnalités précédemment lancées.

Par rapport à la représentation classique d'un composant, chapitre 4, section 4.4 :

- Les méthodes de configuration représentent les propriétés de configuration
- Les méthodes d'entrée et de sortie représentent les interactions possibles avec le composant (interfaces d'entrée).
- Les interfaces requises ne sont pas prises en compte pour le moment dans notre solution. Aujourd'hui, les composants peuvent utiliser d'autres composants. Ils dépendent alors d'eux. Pour autant aucun lien n'a été clairement spécifié. L'intégration d'un

composant dans un autre composant se fait avec notre solution. Par la suite, nous pourrons mettre en place ce système.

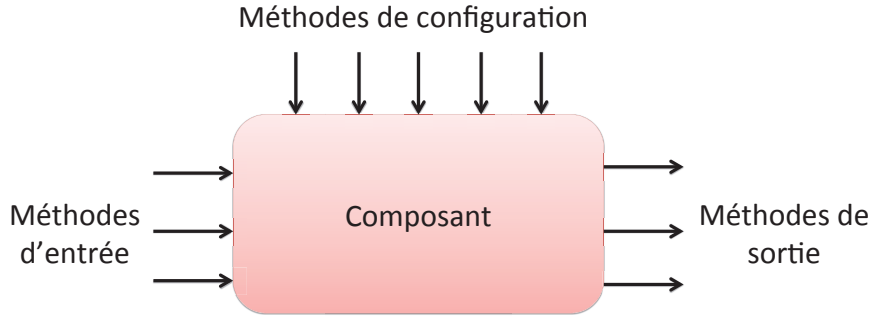


FIGURE 6.1 – Entrée/sortie d'un composant

En terme de programmation, n'importe quelle méthode (d'entrée ou de configuration) sera caractérisée par une méthode que l'intégrateur pourra appeler après l'instanciation du composant. Pour rappel, chacune des méthodes de nos composants pourra être appelée avec notre langage commun. Nous reviendrons sur ce point dans le prochain chapitre.

Nos composants doivent être capables de fonctionner sous deux modes distincts : soit ils fonctionnent en mode synchrone, soit en mode asynchrone. En effet, il est très important dans le développement mobile de pouvoir lancer des processus en mode asynchrone. Sans ça, les applications mobiles seraientt bloquées trop souvent. Sur les figures 6.2 et 6.3, nous représentons les deux processus. Sur la figure 6.2, nous voyons que l'utilisateur de l'application est bloqué pendant le traitement. Si le traitement est trop long, plusieurs secondes, l'utilisateur risque de ne pas accepter cette attente. Par conséquent il risque de rapidement désinstaller l'application. À partir de ce constat, une des règles pour garantir une expérience utilisateur optimale est de ne jamais bloquer l'utilisateur avec un traitement lourd. Sur la figure 6.3, nous voyons qu'en mode asynchrone nos composants doivent être capables d'appeler l'application hôte pour notifier la fin de leurs traitements et retourner les résultats souhaités. Pour ce faire, il faut que l'application hôte implémente certaines méthodes : ce sont les méthodes de sortie de nos composants.

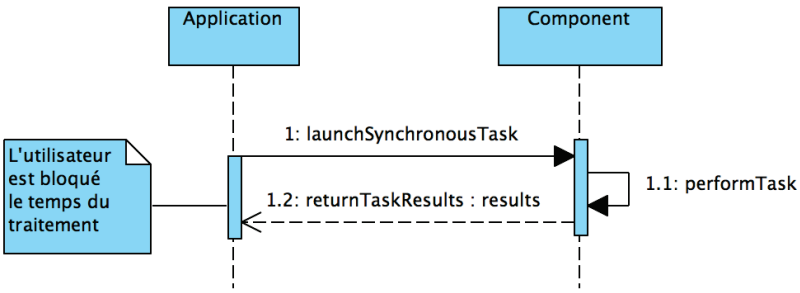


FIGURE 6.2 – Processus synchrone

6.2.2 La délégation : un moyen de rendre un composant indépendant de toutes les applications

Pour que nos composants soient utilisés fréquemment par les utilisateurs de notre solution, il doivent être implémentés de façon générique et non pour une application particulière. Cela implique qu'ils ne peuvent pas faire de traitements spécifiques à une application. Pour mettre en oeuvre ce principe, nous intégrons le patron de conception delegate. Il est basé sur les

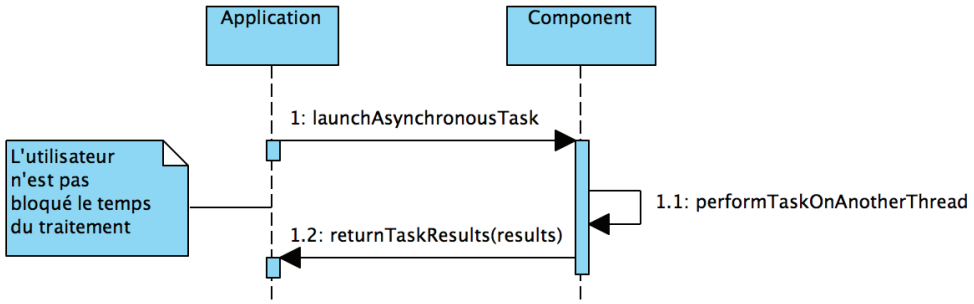


FIGURE 6.3 – Processus asynchrone

principe d'inversion de contrôle [JF88]. Ce patron de conception permet en effet de déléguer certains traitements à une autre entité logicielle. Dans notre cas, nous déléguerons tous les traitements spécifiques à l'application concernée.

Prenons un exemple concret, à keyneosoft, nous sommes spécialistes dans le m-commerce et parmi les produits que nous présentons nous avons un logiciel de gestion de liste d'achats cross-canal. Avec ce produit, un utilisateur peut enregistrer sa liste d'achats à partir d'un site web et la récupérer sur son smartphone (Android, iOS ...) pour aller faire ses courses. Pour arriver à ce résultat, ce logiciel est divisé en trois parties : un site web, un serveur et un composant multiplateforme côté mobile. Le site web permet de faire des recherches dans la liste des produits disponibles sur le serveur, de créer une liste d'achats et de la remplir. Pour cela, le serveur expose plusieurs web services qui s'occupent de gérer les données. Il expose aussi un web service qui permet de récupérer une liste d'achats à partir d'un identifiant. Ce web service est principalement consommé côté mobile par le composant multiplateforme "ShoppingList". Ce composant permet, en plus de récupérer une liste d'achats du serveur, de l'enregistrer localement sur le smartphone, de valider ou non l'achat d'un produit ... Ici, nous n'allons pas détailler plus ce logiciel. Ce n'est pas l'intérêt de la thèse. Nous allons uniquement nous focaliser sur la récupération d'une liste d'achats à partir du composant "ShoppingList".

En fonction du magasin dans lequel le logiciel fonctionne, la base de données produits peut changer. Par conséquent, la définition d'un produit peut changer. Pour certains magasins nous aurons pour chaque produit un champ poids, alors que pour un autre, nous ne l'auront pas. Cela engendre des différences dans la récupération d'une liste d'achats d'un magasin à un autre. En effet, le web service qui permet la récupération d'une liste d'achats retourne la liste de produits sous forme JSON. C'est ici qu'il peut y avoir des spécificités entre les différents magasins et donc dans le traitement dans la récupération de la liste. En fonction du magasin, la récupération et le parsing d'un produit pourront être différents.

Ces différences de définition de produits ne doivent surtout pas être traitées par le composant. Dans ce cas, nous devrions créer un composant "ShoppingList" par magasin cible et donc par application. Au lieu de faire ceci, pour chaque produit trouvé dans la réponse du web service, nous déléguons son parsing à l'application hôte. Sur le diagramme de séquence 6.4, nous représentons ce processus. Nous remarquons que le composant interagit avec l'application pendant son traitement.

Avec cet exemple, nous montrons que le système de délégation est un bon moyen de rendre un composant complètement indépendant d'une application particulière. Ce système est d'ailleurs très fréquemment utilisé. Par exemple, Apple l'intègre dans beaucoup des classes du SDK Mac OS et iOS [BFL11]. Dans notre solution, les méthodes appelées dans le cadre de la délégation sont considérées comme des méthodes de sortie au même titre qu'une méthode de sortie classique, c'est-à-dire, une méthode appelée à la fin du traitement du composant.

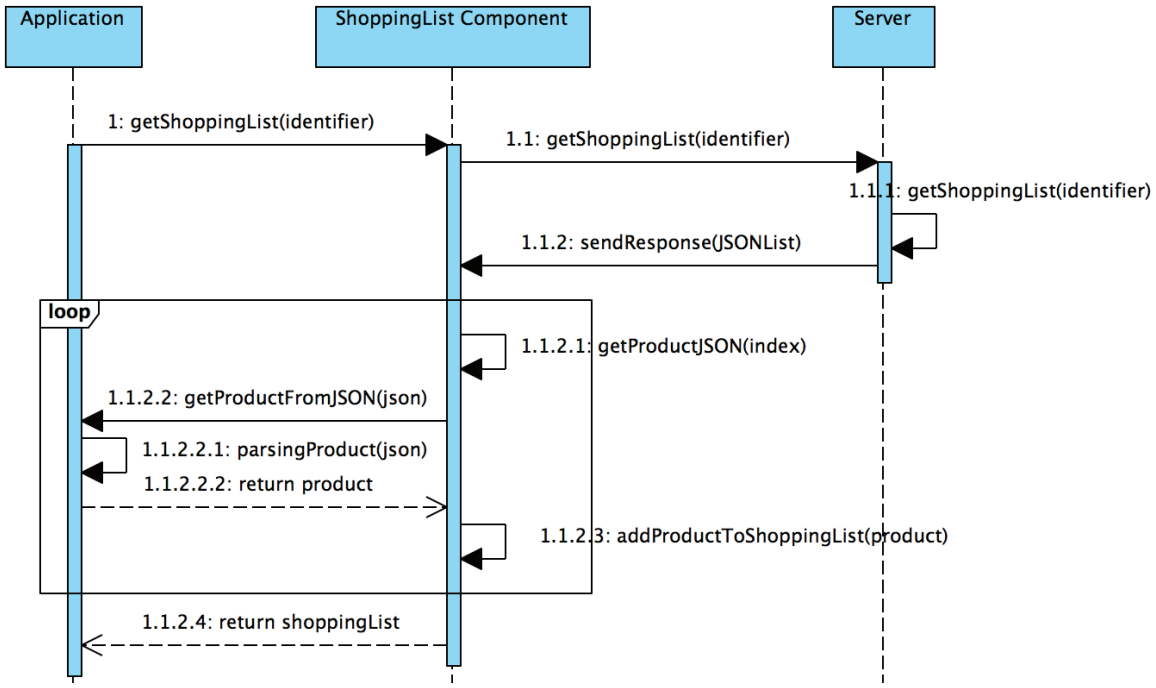


FIGURE 6.4 – Interactions entre une application et le composant "ShoppingList"

Pour faciliter l'utilisation de nos composants, toutes les méthodes de sortie (appelées au milieu d'un traitement ou à la fin) seront implémentées à travers un delegate.

Pour récapituler, les intégrateurs peuvent paramétrer un composant à partir des méthodes de configuration. Il pourront ensuite lancer les méthodes d'entrée de celui-ci de façon synchrone ou asynchrone. Enfin, tout ce qui sera spécifique à une application sera géré par un système de délégation. Ainsi, nos composants sont complètement indépendants d'une application particulière.

6.2.3 Un exemple concret : "httpRequestManager"

Tout au long de cette thèse, dans le but d'illustrer nos propos, nous allons prendre un exemple concret de composant et donner toutes les phases nécessaires pour arriver à son implantation et son utilisation dans notre framework de développement multiplateforme. Nous allons nous focaliser en particulier sur le composant "HttpRequestManager". Ce composant permet d'appeler n'importe quel web service à partir d'une requête HTTP. Il est très utilisé à Keyneosoft lors du développement de nos applications. Il est très rare de faire une application mobile sans aucun appel à des Web Services. De plus, il est assez représentatif de tout ce que nous voulons montrer dans cette thèse. En effet, le composant fonctionne en mode asynchrone. Il délègue le traitement des données reçues et se présente sous la forme d'un singleton.

La première phase du développement d'un composant consiste à définir les entrées et sorties de celui-ci. Pour ce faire, le développeur de composants doit le concevoir en faisant abstraction des plate-formes cibles. Il doit lister les méthodes d'entrée et de configuration qu'il veut exposer.

Le composant "HttpRequestManager" a plusieurs fonctionnalités (méthodes d'entrée), figure 6.6 :

— **Envoyer une requête Http (de façon asynchrone) :** avant de lancer la requête

le composant vérifie que le téléphone soit connecté à internet. Si ce n'est pas le cas, il notifie l'application hôte qu'il n'y a pas de réseau. Si c'est le cas, le composant s'occupe de créer un client Http avec le serveur correspondant et de récupérer le résultat. Si le composant à déjà des requêtes en cours d'exécution, il exécute celle-ci en parallèle. Lorsque la réponse est téléchargée, il notifie l'application hôte et lui délègue le traitement des données. Dans le cas où une erreur survient (status différent de 200 par exemple), le composant notifie l'application hôte. Ce processus correspond à la méthode ***sendHttpRequest(request)*** ;

— **Annuler toutes les requêtes en cours :** le composant s'occupe d'annuler toutes les requêtes en cours. Il notifie l'application hôte pour chaque requête annulée. Ce processus correspond à la méthode ***cancelAllHttpRequests()*** ;

— **Annuler une requête particulière :** le composant s'occupe d'annuler la requête, il notifie l'application hôte de son annulation. Ce processus correspond à la méthode ***cancelHttpRequest(request)***.

Ensuite, le développeur de composants doit définir les méthodes de sortie. Avec ces méthodes, le composant peut déléguer certains traitements et notifier l'application hôte :

— **De la fin des traitements d'une requêtes particulières :** le composant retourne alors les données texte correspondantes à la requête souhaitée : ***textReceived(text, request)***.Pour des raisons de lisibilité, nous nous limitons aux données de type texte. En réalité, le composant gère beaucoup plus de données que cela.

— **D'une erreur :** il est possible que le composant ne puisse pas récupérer le résultat de la requête souhaitée (pas de connexion internet, problème serveur ...). Dans ce cas, il retourne une erreur : ***serviceFailed(error, request)***.

— **Qu'une requête a été annulée :** si l'intégrateur annule une requête ou toutes les requêtes, nous notifions à l'application des requêtes annulées : ***requestCancelled(request)***. La méthode prend en entrée la requête annulée.

Toutes les méthodes de sortie sont regroupées dans un delegate que l'application hôte doit implémenter : HttpRequestManagerDelegate.

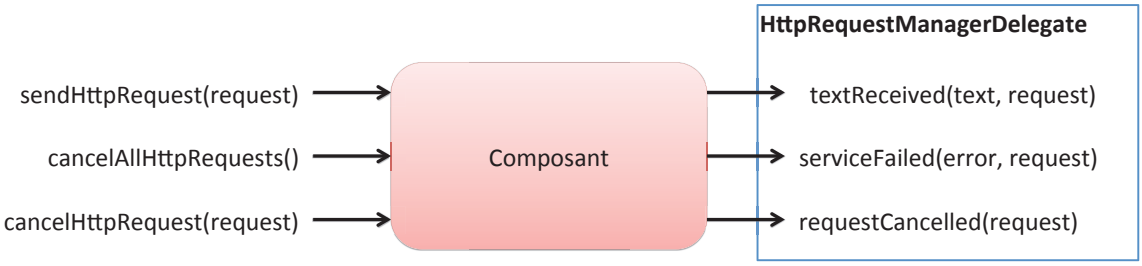


FIGURE 6.5 – Entrées/sorties du composant "HttpRequestManager"

Sur la figure 6.6, nous présentons un des processus possible lors de l'utilisation du composant. Nous montrons que l'application hôte appelle la méthode d'entrée sendHttpRequest. Ensuite, lorsque le composant a fait son traitement, il retourne le résultat avec la méthode de sortie dataReceived. C'est l'application hôte qui doit donc implémenter cette méthode. Nous voyons aussi que le composant délègue le traitement des données reçues à l'application. En effet, le composant "httpRequestManager" est souvent imbriqué dans un processus plus complet. Le composant est utilisé pour appeler un web service. Cependant il n'est pas capable de traiter toutes les réponses possibles (json, xml ...) sans connaître les formats de données retournées par ce web service. Nous supposons que l'utilisateur du composant, lui, a connaissance de ces formats et que par conséquent c'est à lui de parser et traiter les données reçues. Le composant ne s'occupe donc que d'une partie du traitement, la récupération des

données, et ensuite délègue le traitement de la réponse à l'application hôte.

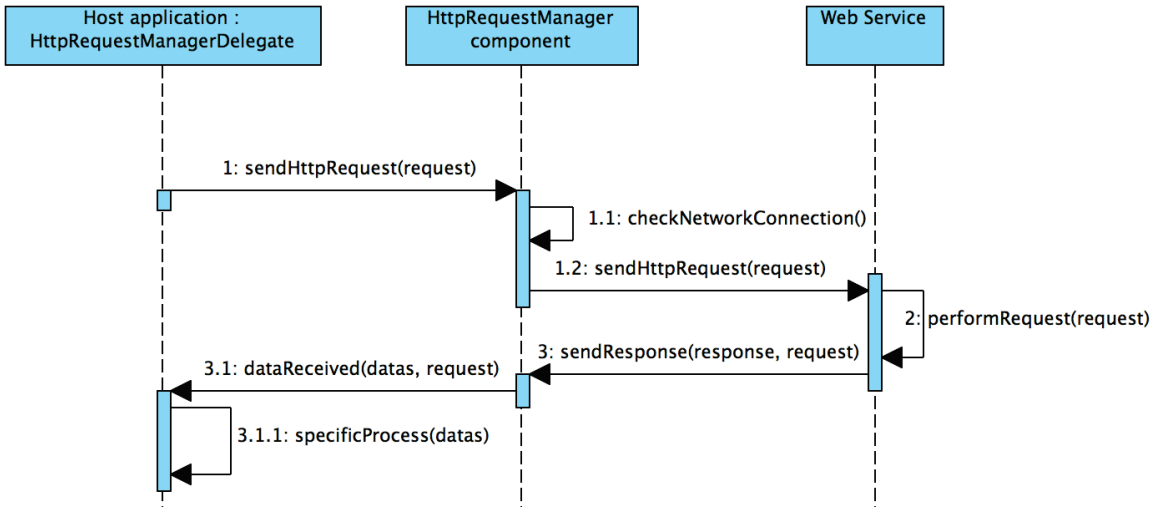


FIGURE 6.6 – Interactions entre une application et le composant "HttpRequestManager"

Par la suite, nous allons étudier la structure interne de nos composants. Nous allons nous focaliser sur leurs implémentations et la façon dont nous exposons leurs méthodes d'entrée, de configuration et de sortie. Pour illustrer nos propos, nous continuerons à prendre comme exemple le composant "HttpRequestManager".

6.3 Structure interne d'un composant multiplateforme

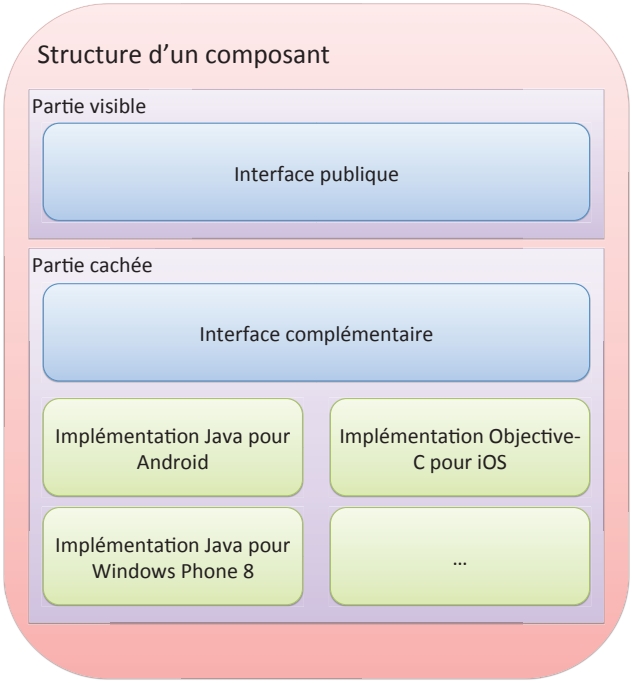


FIGURE 6.7 – Structure d'un composant multiplateforme dans notre solution

Sur la figure 6.7, nous représentons la structure interne de nos composants, c'est-à-dire, tous les éléments contenus dans un composant multiplateforme. Ce type de composant est divisé en deux parties :

— **La partie cachée :** elle représente l'implantation du composant. Par rapport aux composants classiques, ce type de composant a la particularité d'avoir une implémentation par plate-forme cible et une interface dite complémentaire. Cette partie est complètement cachée aux développeurs utilisant notre solution car elle spécifique à chacune des plateformes cibles. En effet, chacune des implémentations du composant est développée pour un environnement d'exécution spécifique et nous verrons que l'interface complémentaire représente toutes ces implémentations.
— **La partie visible :** elle représente les fonctionnalités du composant tout en étant complètement indépendante de chacune des plates-formes cibles. Cette partie contient une interface publique écrite en XML. C'est à partir de cette interface que les développeurs, les utilisateurs de notre solution pourront intégrer des composants en utilisant notre langage commun. L'interface reprend les méthodes d'entrée, de configuration et de sortie du composant présentées sur la figure 6.1.

Par la suite, nous allons étudier les deux parties de la structure d'un composant.

6.4 Description de la partie cachée d'un composant multiplateforme

Dans cette section, nous nous focalisons sur la partie cachée du composant qui contient deux couches distinctes : une implémentation native par plate-forme et une interface complémentaire.

6.4.1 Une implémentation par plate-forme cible

Dans notre solution, un composant a une implémentation spécifique à chaque plate-forme cible. Chaque implémentation est développée de façon native. En effet, elles sont développées avec les outils et langages de programmation officiels. En utilisant cette stratégie, nous répondons à nos attentes qui sont :

— Fournir une expérience utilisateur identique au natif (réactive et fluide).
— Garder l'intégrité de chaque plate-forme cible.
— Fournir une application composée uniquement de code natif.

Effectivement, puisque nous ne passons pas par d'autres environnements d'exécution, nous nous assurons que les composants fournissent les meilleures performances possibles et donc également la meilleure expérience possible. Les solutions comme PhoneGap, par exemple, qui exécutent du code à la volée ne permettent pas d'obtenir des performances satisfaisantes [CSS12]. Dans les chapitres 9 et 10, nous avons comparé ce genre de solution avec la notre et le développement natif. Nous avons constaté que les applications générées avec notre solution sont aussi réactives et fluides que lorsqu'elles sont développées complètement nativement.

De plus, nos composants pourront s'intégrer parfaitement dans une application mobile native et un écosystème natif. Si nous reprenons l'exemple du chapitre 5 dans l'introduction sur le choix d'une date ou l'intégration d'une carte à partir d'un composant sur différentes plates-formes, nous voulons mettre en avant les différences de chaque plate-forme. Avec notre solution, nous pourrons implémenter des composants, qui en fonction de la plate-forme utilisée,seront des moyens différents pour effectuer leurs fonctions. Ainsi, nous gardons l'intégrité de chaque plate-forme cible. C'est un point clé de notre solution, sans ça, la même application sera identique sur iOS, Android, etc. ce que nous ne voulons surtout pas par rapport à nos attentes.

Enfin, comme nous l'avons montré dans le chapitre 5, section 5.3, notre compilateur sélectionnera l'implémentation du composant à intégrer dans les applications générées en fonction de la plate-forme cible. Les applications générées ne contiennent alors que du code source natif.

6.4.2 Une interface complémentaire

En plus de l'implémentation qui est cachée, nous dissimulons une interface qui est dite "complémentaire". Cette interface représente les fonctionnalités exposées par chacune des implémentations natives du composant (méthodes d'entrée, de configuration et de sortie). Pour cela, elle reprend toutes les méthodes exposées par le composant pour les retranscrire en XML. Nous avons choisi d'utiliser le XML pour cette interface car c'est un langage suffisamment expressif pour ne pas être limité dans la re-transcription de chaque méthodes visible du composant. Il est d'ailleurs souvent utilisé pour décrire des composants. C'est le cas dans les WSDL. De plus, ce langage peut être compris par tous les développeurs mobiles. Ce langage est, en effet, utilisé très fréquemment dans le mobile. Il sert notamment à implémenter l'interface graphique des vues sur Android. Il est aussi très utilisé dans les réponses des web services. Il permet ainsi de formater les données retournées. Nous donnerons un exemple concret de cette interface dans la section 6.4.3. L'utilisation de cette interface sera complètement transparente pour les développeurs utilisant notre solution. Elle est, en effet, principalement utilisée par nos outils de compilation lors de la transformation de notre langage commun en langage natif. Nous reviendrons sur ce point dans le prochain chapitre.

À travers la figure 6.8, nous montrons la hiérarchie que l'interface complémentaire doit suivre. Elle est divisée en trois parties distinctes :

- La **description** du composant (*1 à 1*) avec son nom, sa version, sa date de génération et une description.
- Les **méthodes de configuration** du composant (*0 à n*). Pour chacune d'elles, nous aurons une (ou plusieurs) description native de la méthode pour chaque plate-forme cible. Chaque méthode native est différenciée par son système d'exploitation, le nom de la classe dans laquelle elle est implémentée, le nom de la méthode et ses paramètres d'entrée (*0 à n*) et de sortie (*0 ou 1*).
- Les **méthodes d'entrée** du composant (*1 à n*). Il faut qu'il y ait au moins une méthode d'entrée dans notre modèle. Sans ça le composant n'est pas intégrable. Comme pour les méthodes de configuration, pour chaque méthode d'entrée définie de façon indépendante, nous aurons au moins une description native de la méthode pour chaque plate-forme cible (*1 à n*). En plus d'une description classique des méthodes natives, il est possible pour chacune d'elle d'associer des **méthodes de sortie** via des delegates (*0 à n*).

Pour récapituler, les implémentations et l'interface complémentaire des composants sont complètement dépendantes de toutes les plates-formes cibles. C'est pour cela que nous la cachons aux intégrateurs de nos composants.

6.4.3 La partie cachée du composant "HttpRequestManager"

Dans la section 6.2.3, nous avons présenté les fonctionnalités du composant "HttpRequestManager" de façon indépendante de toutes les plates-formes cibles. Dans cette section,

extraInformation ----------(1..1)
| **description** ----------(1..1)
| | **componentName** ---------- (1..1)
| | **version** ---------- (1..1)
| | **date**---------- (1..1)
| | **description** ----------(1..1)
| **configurations** ---------- (0..1)
| | **extraInformationOnConfiguration***(name)* ---------- (0..*n*)
| | | **android** *(class, package, isSingleton, name)* ----------(0..*n*)
| | | | **Parameter** *(identifier, type, io, functional)* ---------- (0..*n*)
| | | **iOS** *(class, isSingleton, name)* ---------- (0..*n*)
| | | | **Parameter**(*identifier, type, io, functional, argument)* ------- (0..*n*)
| **methods** ---------- (1..1)
| | **extraInformationOnMethod***(name)* ----------(1..*n*)
| | | **android** *(class, , isSingleton, name)* ----------(0..*n*)
| | | | **parameter***(identifier, type, io, functional)* ---------- (0..*n*)
| | | | **delegate** *(interface, package)*---------- (0..*n*)
| | | | | **callbackMethod***(name)* ---------- (1..*n*)
| | | | | | **parameter***(identifier, type, io, functional)* ---------- (0..*n*)
| | | **iOS** *(class, isSingleton, name)* ---------- (0..*n*)
| | | | **parameter** *(identifier, type, io, functional, argument)* ------ (0..*n*)
| | | | **delegate** *(protocol)* ---------- (0..*n*)
| | | | | **callbackMethod** *(name)* ---------- (1..*n*)
| | | | | | **parameter** *(identifier, type, io, functional, argument)* (0..*n*)

FIGURE 6.8 – Hiérarchie de l'interface complémentaire d'un composant

nous allons montrer comment le développeur de composants les a implémenté nativement pour chacune des plates-formes cibles (Android et iOS). Au niveau de la programmation, chaque fonctionnalité du composant est représentée par une méthode native. Avec les codes sources 6.1 et 6.2, nous avons défini une interface native pour Android puis pour iOS. Ces interfaces représentant les méthodes d'entrée du composant.

Code 6.1 – Interface native pour Android du composant HttpRequestManager

```
1  public interface HttpRequestServiceInterface
2  {
3      public void sendHttpGetRequest(Context context,
4              HttpGet getRequest,
5              HttpGetRequestServiceDelegate delegate);
6
7      public void sendHttpPostRequest(Context context,
8              HttpPost postRequest,
9              HttpPostRequestServiceDelegate delegate);
10
11     public void cancelAllHttpRequests();
12
13     public void cancelHttpGetRequest(HttpGet request);
14
15     public void cancelHttpPostRequest(HttpPost request);
16 }
```

Code 6.2 – Interface native pour iOS du composant HttpRequestManager

```
1  @interface HttpRequest : NSObject
2
3  +(HttpRequest*)sharedInstance;
4
```

```
5 -(void)sendHttpRequest:(NSURLRequest*)request
6                  delegate:(id<HttpRequestDelegate>)delegate;
7
8 -(void)cancelAllHttpRequests;
9
10 -(void)cancelHttpRequest:(NSURLRequest*)request;
11
12 @end
```

Nous remarquons plusieurs différences :

— Il y a plus de méthodes sur Android que de fonctionnalités. Cela s'explique par le fait que le composant peut envoyer n'importe quel type de requête HTTP [1]. Sur Android, chaque type de requête HTTP est spécifié par un objet différent : HttpGet pour les requêtes de type Get et HttpPost pour les requêtes de type POST. Dans cette thèse, nous nous limitons à ces deux types pour des raisons de lisibilité. Pour gérer ces types différents, nous avons doublé le nombre de méthodes qui attendaient en entrée une requête HTTP.

— Les méthodes liées à l'envoi des requêtes sur Android prennent en entrée un objet de type Context. Cet objet est complètement lié à l'environnement Android. Il n'y a d'ailleurs aucun équivalent sur iOS ou sur une autre plate-forme mobile.

— Les langages de programmation sont différents.

— Les types des objets passés en entrée des méthodes sont différents.

Nous verrons dans la suite comment nous cachons ces différences à l'utilisateur, c'est-à-dire, l'intégrateur de composants.

Nous avons effectué le même traitement avec les méthodes de sortie du composant. Ainsi, sur les codes sources 6.3 et 6.4, nous avons représenté les méthodes de sortie dans un delegate. Comme pour les méthodes d'entrée, il y a quelques différences. Nous avons divisé la méthode de sortie ***serviceFailed(error,request)*** en plusieurs méthodes, une par type d'erreur possible. De plus, classiquement, un delegate sur Android est représenté par une interface Java alors que pour iOS il est représenté par un protocole. Il y a aussi des différences entre les types des objets retournés.

Code 6.3 – Représentation du delegate HttpRequestDelegate en Java pour Android

```
1 public interface HttpGetRequestServiceDelegate {
2
3     public void textRecieved(String text, HttpResponse response, HttpGet
          request);
4
5     public void requestCancelled(HttpGet request);
6
7     public void requestError(Exception error, HttpGet request);
8
9     public void statusError(HttpResponse response, HttpGet request);
10
11    public void noActiveNetworkConnection(HttpGet request);
12 }
```

Code 6.4 – Représentation du delegate HttpRequestDelegate en Objective-C pour iOS

```
1 @protocol HttpRequestDelegate <NSObject>
2
```

1. Les requêtes HTTP peuvent permettre plusieurs actions. Chaque action est caractérisée par un type de requête. Ainsi, il existe des requêtes HTTP de type Get (demande au serveur une ressource, Post (transmet des données au serveur), Head (demande des informations sur une ressource) ...

```
3  -(void)textReceived:(NSString*)text
4            responding:(NSHTTPURLResponse*)response
5                    to:(NSURLRequest*)request;
6
7  -(void)requestCancelled:(NSURLRequest*)request;
8
9  -(void)requestError:(NSError*)error
10                   to:(NSURLRequest*)request;
11
12 -(void)statusError:(NSHTTPURLResponse*)response
13                  to:(NSURLRequest*)request;
14
15 -(void)noActiveNetworkConnection:(NSURLRequest*)request;
16
17 @end
```

Après avoir défini les méthodes d'entrée, de configuration et de sortie en langage natif pour chacune des plates-formes cibles, le développeur de composants peut implémenter son composant. Pour gérer plusieurs appels aux web services de façon concurrentes, sur iOS, nous utilisons un objet de type NSOperationQueue. Sur Android, nous utilisons une simple liste de type ArrayList qui contiendra l'état du composant avec toutes les requêtes en cours. Ensuite, sur iOS, la gestion du mode asynchrone se fait automatiquement dans l'objet de type NSOperationQueue alors que sur Android nous devons créer notre propre classe qui hérite de la classe AsyncTask. Ces deux procédés sont radicalement différents et seront cachés pour l'utilisateur final. Toutes ces différences seront donc transparentes pour l'utilisateur. Il va uniquement appeler les méthodes publiques du composant (envoyer une requête, annuler une requête) sans avoir le détail du processus.

De plus, il y a une différence de comportement au niveau ergonomie du composant sur iOS par rapport à Android. En effet, Apple préconise d'afficher dans la barre de statut d'iOS un indicateur d'activité de réseau. Tant qu'il y a des appels à des web services en cours, nous affichons cet indicateur comme il est montré sur la figure 6.9. Sur Android, cet élément n'existe pas. Toutes ces différences seront donc cachées à l'utilisateur de notre solution ce qui lui évitera de se former à ce processus et surtout d'oublier certaines parties. Il n'est pas rare qu'un développeur Android qui se forme ensuite sur iOS, ne gère pas du tout l'indicateur d'activité réseau sur iOS ce qui en fin de compte diminue l'expérience utilisateur. L'utilisation d'un composant augmente souvent la qualité des logiciels les utilisant. Toutes ces différences seront complètement transparentes pour l'utilisateur de ce composant.

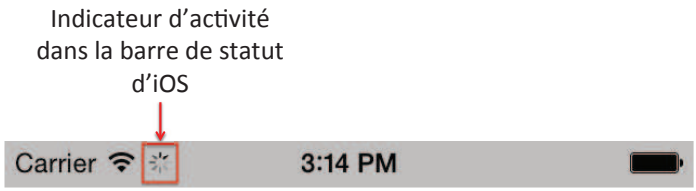


FIGURE 6.9 – Affichage d'un indicateur d'activité dans la barre de statut d'iOS lorsqu'on effectue une connexion internet

Maintenant, voyons comment passer des interfaces natives du composant à l'interface complémentaire de celui-ci. L'interface complémentaire reprend toutes les méthodes natives de toutes les implémentations du composant en XML. Avant de commencer ce traitement, nous devons réunir les méthodes natives sous des fonctionnalités communes. Sur la figure 6.10, nous avons regroupé les méthodes en suivant les fonctionnalités que nous avions décrites de façon indépendante d'une plate-forme cible. En bleu, nous regroupons les méthodes liées à

```
public interface HttpRequestServiceInterface
{
    public void sendHttpGetRequest(Context context,
            HttpGet getRequest,
            HttpGetRequestServiceDelegate delegate);

    public void sendHttpPostRequest(Context context,
            HttpPost postRequest,
            HttpPostRequestServiceDelegate delegate);

    public void cancelAllHttpRequests();

    public void cancelHttpGetRequest(HttpGet request);

    public void cancelHttpPostRequest(HttpPost request);
}
```

```
@interface HttpRequest : NSObject

+(HttpRequest*)sharedInstance;

-(void)sendHttpRequest:(NSURLRequest*)request
              delegate:(id<HttpRequestDelegate>)delegate;

-(void)cancelAllHttpRequests;

-(void)cancelHttpRequest:(NSURLRequest*)request;

@end
```

FIGURE 6.10 – Réunion des méthodes natives par fonctionnalités

l'envoi d'une requête HTTP, en rouge, l'annulation de toutes les requêtes HTTP en cours et en vert l'annulation d'une requête HTTP particulière. Nous pouvons donc avoir plusieurs méthodes natives pour la même méthode d'entrée.

Avec le code 6.5, nous donnons une partie de l'interface complémentaire du composant "HttpRequestManager". Pour simplifier la lecture, nous avons délibérément caché certaines parties qui étaient redondantes. Nous listons donc toutes les méthodes d'entrée et de sortie du composant pour chacune des plates-formes cibles sans cacher les spécificités de chacune d'elle. C'est pour cela que, finalement, nous cachons cette interface aux intégrateurs de composants.

Code 6.5 – Interface complémentaire du composant "HttpRequestManager"

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <extraInformation xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:noNamespaceSchemaLocation="ComplementInterfaceSchema.xsd" >
4
5      <description>
6          <componentName>HttpRequestManager</componentName>
7          <version>1.0</version>
8          <date>10/05/2013</date>
9          <description>...</description>
10     </description>
11
12     <methods>
13         <extraInformationOnMethod name="sendHttpRequest">
14             <android class="HttpRequestService"
15                 package="com.keyneosoft.requestManager.implementations"
16                 isSingleton="true"
17                 name="sendHttpGetRequest">
18                 <parameter identifier="context" type="Context" io="in"
                       functional="false"/>
19                 <parameter identifier="getRequest" type="HttpGet" io="in"
                       functional="true"/>
20                 <parameter identifier="delegate" type="
                       HttpGetRequestServiceDelegate" io="in" functional="true
                       "/>
21                 <delegate interface="HttpGetRequestServiceDelegate"
22                     package="com.keyneosoft.requestManager.delegates">
23                     <callbackMethod name="textReceived">
24                         <parameter identifier="text" type="String" io="in"
                               functional="true"/>
25                         <parameter identifier="response" type="HttpResponse
                               " io="in" functional="true"/>
26                         <parameter identifier="request" type="HttpGet" io="
                               in" functional="true"/>
27                     </callbackMethod>
```

```
28                      <callbackMethod ...>
29                  </delegate>
30              </android>
31
32              <android class="HttpRequestService" package="com.keyneosoft.
                    requestManager.implementations"
33                  isSingleton="true">
34                  <parameter identifier="context" type="Context" io="in"
                        functional="false"/>
35                  <parameter identifier="postRequest" type="HttpPost" io="in"
                         functional="true"/>
36                  <parameter identifier="delegate" type="
                        HttpPostRequestServiceDelegate" io="in" functional="
                        true"/>
37                  <delegate ...>
38              </android>
39
40              <iOS class="HttpRequest" isSingleton="true" method="
                    sendRequest:delegate:andActivityIndicatorVisible:">
41                  <parameter identifier="request" type="NSURLRequest*" io="in
                        " argument="sendRequest" functional="true"/>
42                  <parameter identifier="delegate" type="
                        idHttpRequestDelegate" io="in" argument="delegate"
                        functional="true"/>
43                  <delegate protocol="HttpRequestDelegate">
44                      <callbackMethod name="textReceived:responding:to:">
45                          <parameter identifier="text" type="NSString*" io="
                                in" argument="textReceived" description="
                                Documentation" functional="true"/>
46                          <parameter identifier="response" type="
                                NSHTTPURLResponse*" io="in" argument="
                                responding" description="Documentation"
                                functional="true"/>
47                          <parameter identifier="request" type="NSURLRequest*
                                " io="in" argument="to" description="
                                Documentation" functional="true"/>
48                      </callbackMethod>
49                      <callbackMethod ...>
50                  </delegate>
51              </iOS>
52          </extraInformationOnMethod>
53
54          <extraInformationOnMethod method="cancelAllHttpRequests" ...>
55
56          <extraInformationOnMethod method="cancelRequest" ...>
57      </methods>
58  </extraInformation>
```

6.5 Description visible d'un composant multiplateforme

Dans cette section, nous cachons toutes les différences soulignées précédemment entre les implémentations natives à travers une interface publique indépendante de toutes les plates-formes mobiles.

6.5.1 Une interface publique

La partie visible du composant est constituée d'une seule interface dite publique. Cette interface représente les fonctionnalités du composant de façon complètement indépendante d'une plateforme cible particulière. Ainsi, à partir de n'importe quel environnement de développement natif, il est possible d'appeler une des fonctionnalités du composant. C'est ce qui permet à un utilisateur de notre solution d'écrire une seule fois le code source avec notre langage universel et de le réutiliser sur toute les plates-formes cibles. Comme nous l'avons relevé précédemment, notre structure de composants pose une nouvelle problématique. Nous devons être capables de réunir plusieurs fonctionnalités provenant de plusieurs implémentations. Pour réussir ce challenge, nous avons défini deux règles à suivre :

— Si une fonctionnalité ne peut pas être disponible pour une plate-forme particulière, cette fonctionnalité est exclue pour le système d'exploitation. Elle n'apparaitra pas dans l'interface publique pour ce système d'exploitation en particulier.
— Toutes les spécificités d'une plate-forme cible doivent être bannies de l'interface publique.

La première règle est facile à mettre en oeuvre. Dans l'interface publique, pour chaque fonctionnalité, nous fournissons une liste des systèmes d'exploitation la supportant. Il sera alors impossible pour un utilisateur d'invoquer une fonctionnalité sur un système d'exploitation qui n'est pas dans sa liste. Si l'utilisateur le fait, notre compilateur lui remontra une erreur.

La deuxième règle est la plus complexe à appliquer. Nous devons cacher toutes les spécificités de chaque plate-forme cible (types différents des objets, paramètres spécifiques à une plate-forme ...). Par conséquent, l'interface publique ne contient aucune référence à des éléments natifs. Les types des variables ne sont donc pas intégré dans l'interface publique. Par exemple, une chaine de caractères qui est de type String en Java et C#, NSString en Objective-C ne sera pas différenciée dans l'interface publique. Elle ne contient pas non plus les paramètres non fonctionnels, c'est-à-dire, les paramètres qui ne servent pas réellement à la fonctionnalité appelée. Par exemple, sur Android, plusieurs processus ont besoin d'un contexte (type Classe) pour être exécutés. De la même façon, sur iOS, lorsque nous voulons ouvrir une nouvelle vue, nous devons passer en entrée un bundle qui correspond à l'endroit où est localisée la ressource demandée, ici la vue. Ce type de paramètres correspond à des spécificités liées à l'environnement de développement de la plate-forme cible. Nous ne les ajoutons donc pas dans l'interface publique de nos composants.

À travers la figure 6.11, nous présentons la structure de l'interface publique d'un composant. Comme pour l'interface complémentaire, nous avons choisi le XML car il est suffisamment expressif pour répondre à nos besoins. L'interface est divisée en quatre parties :

— La **description** du composant (*1 à 1*) avec son nom, sa version, sa date de génération et une description.
— Les **plates-formes cibles** du composant (*1 à n*). Ici, nous listons toutes les plateformes sur lesquelles le composant peut être déployé. Le composant doit être au moins déplorable sur une plate-forme. Pour chacune des plates-formes cibles, nous lui donnons son nom et le nom de l'exécutable à intégrer dans l'application l'utilisant.
— Les **méthodes de configuration** du composant (*0 à n*). Chaque méthode est décrite de façon indépendante d'une plate-forme cible. Pour ce faire, contrairement à l'interface complémentaire, nous n'intégrons plus le type des paramètres. Nous ne faisons plus la distinction entre les méthodes natives. Enfin, pour chaque méthode, nous listons les plates-formes sur lesquelles elle peut être utilisée. Nous verrons dans les prochains chapitres qu'il se peut qu'une méthode particulière ne soit pas disponible pour

component ---------- (1..1)
| **description** ---------- (1..1)
| | **componentName** ---------- (1..1)
| | **version** ---------- (1..1)
| | **date**---------- (1..1)
| | **description** ----------(1..1)
| **targetPlatforms** ---------- (1..1)
| | **target***(name, executable)*---------- (1..*n*)
| **configurations** ---------- (0..1)
| | **method***(name, description)* ---------- (0..*n*)
| | | **parameter***(identifier, io, description)* ----------(0..*n*)
| | | **target***(name)* ----------(1..*n*)
| **methods** ----------(1..1)
| | **method***(name, description)* ----------(1..*n*)
| | | **parameter***(identifier, io, description, delegate)* (0..*n*)
| | | **target***(name)* ---------- (1..*n*)
| | | **callbackMethods***(name)* ---------- (0..*n*)
| | | | **callbackMethod***(name)* ---------- (1..*n*)
| | | | | **parameter***(identifier, io, description)* ---- (0..*n*)
| | | | | **target***(name)* ---------- (1..*n*)

FIGURE 6.11 – Hiérarchie de l'interface commune d'un composant

un système d'exploitation même si le composant peut être utilisé sur la plate-forme en question.

— Les **méthodes d'entrée** du composant (*1 à n*). Comme pour les méthodes de configuration, nous décrivons chaque méthode d'entrée sans faire allusion aux éléments spécifiques des plateformes cibles. Pour chacune des méthodes, nous leurs associons aussi des **méthodes de sortie** qui elles aussi sont décrites de façon indépendante (*0 à n*). Un exemple de cette interface est donné dans la section 6.5.2.

6.5.2 La partie visible du composant "HttpRequestManager"

Pour obtenir l'interface publique du composant "HttpRequestManager", nous repartons de l'interface complémentaire et nous la simplifions pour ne plus avoir de dépendances à une plate-forme cible. Tout d'abord, nous supprimons les paramètres spécifiques à une plate-forme cible et non fonctionnel. Par exemple, sur Android, le composant a besoin d'un objet de type CONTEXT pour fonctionner. Ce paramètre est non-fonctionnels et spécifique à Android. Nous ne l'intégrons donc pas dans l'interface publique. Ensuite, nous supprimons tous les types de toutes les méthodes natives et réunissons toutes les méthodes natives en une seule. Par exemple, nous réunissons les méthode ***sendHttpGetRequest*** et ***sendHttpPostRequest***. Enfin, nous ajoutons la notion de target pour chaque méthode d'entrée, de configuration ou de sortie. Nous obtenons alors l'interface présentée dans le code 6.6.

Code 6.6 – Une partie Interface publique du composant "HttpRequestManager"

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation="CommonInterfaceSchema.xsd" >
4
5     <description>
```

```
6       <componentName>HttpRequestManager</componentName>
7       <version>1.0</version>
8       <date>10/05/2013</date>
9       <description>...</description>
10   </description>
11
12   <targetPlatforms>
13      <target name="android" executable="HttpRequestManager.jar"/>
14      <target name="iOS" executable="HttpRequestManager.framework"/>
15   </targetPlatforms>
16
17   <methods>
18
19      <method name="sendHttpRequest" description="...">
20         <parameter identifier="request" io="in" description="request to
               send"/>
21         <parameter identifier="delegate" io="in" description="method
               delegate" delegate="true"/>
22
23            <callbackMethods>
24
25               <callbackMethod name="textReceived" description="">
26                  <parameter identifier="text" io="in" description="text
                        received"/>
27                  <parameter identifier="response" io="in" description="
                        web service response"/>
28                  <parameter identifier="request" io="in" description="
                        http request"/>
29
30                  <target name="android"/>
31                  <target name="iOS"/>
32               </callbackMethod>
33
34               <callbackMethod ...>
35               </callbackMethod>
36
37            </callbackMethods>
38            <target name="android"/>
39            <target name="iOS"/>
40         </method>
41
42         <method name="cancelAllHttpRequests" description="...">
43            <target name="android"/>
44            <target name="iOS"/>
45         </method>
46
47         <method name="cancelRequest" description="...">
48            <parameter io="in" description="request to cancel "/>
49
50            <target name="android"/>
51            <target name="iOS"/>
52         </method>
53
54
55     </methods>
56
57 </Component>
```

Pour récapituler le composant "HttpRequestManager" est représenté par :

— Une **interface publique** indépendante de toutes les plates-formes cibles. Elle est présentée avec le code 6.6. C'est à partir de cette interface que les intégrateurs du

composant pourront l'intégrer de façon unique dans toutes leurs applications. Ce point sera l'objet du prochain chapitre.
— Une **interface complémentaire** représentant toutes les entrées natives du composant. Elle est présentée avec le code 6.5. Cette interface est cachée aux intégrateurs. Elle servira principalement à nos outils, chapitre 8 qui pourront, grâce à elle, faire le lien entre l'interface publique et les implémentations natives du composant.
— **Plusieurs implémentations natives** du composant représentées par les interfaces natives 6.1 et 6.2.

6.6 Conclusion

Dans ce chapitre, nous avons donné la structure de nos composants multiplateformes. Dans un premier temps, nous avons décrit les interactions possibles entre les intégrateurs et les composants. Elles se divisent en trois catégories : les méthodes de configuration, d'entrée et de sortie. Les intégrateurs de composants pourront paramétrer les composants à partir des méthodes de configuration, appeler les fonctionnalités des composants à partir des méthodes d'entrée et recevoir les résultats à partir des méthodes de sortie. Dans un second temps, nous nous sommes focalisés sur la structure de ce nouveau genre de composants. Contrairement aux composants classiques, ils ont la particularité d'avoir plusieurs implémentations pour plusieurs environnements d'exécutions cibles. Chacune des implémentations est réalisée avec les outils officiels de chacune des plates-formes visées. Par conséquent, elles ont toutes des spécificités propres à chaque environnement cible. Les méthodes exposées par les implémentations natives sont ensuite réunies dans une seule interface dite complémentaire. Les implémentations natives des composants et leurs interfaces complémentaires sont cachées aux utilisateurs. En effet, ces éléments sont complètement dépendants de toutes les plates-formes cibles. Pour dissimuler ces différences, les composants ont une interface dite publique. Cette interface reprend les méthodes de configuration, d'entrée et de sortie du composant de façon indépendante de toutes les plates-formes visées.

En plus de montrer les différents éléments constituant nos composants multiplateformes, nous avons mis en oeuvre tous ces éléments avec un exemple concret : le composant "HttpRequestManager". Ainsi, nous avons décrit toutes les phases à suivre pour créer un composant multiplateforme. Tout d'abord, le développeur de composants doit concevoir son composant de façon multiplateforme à un haut niveau d'abstraction. Il décrit alors les méthodes de configuration, d'entrée et de sortie du composant. Ensuite, pour chaque plate-forme cible, il doit implémenter toutes ces méthodes. Bien sûr, il prendra bien soin de décrire toutes les méthodes dans des interfaces natives. Ce sont ces interfaces qui seront exposées. La façon de les implémenter n'est pas importante pour nous. Après ceci, le développeur peut créer son interface complémentaire. Pour ce faire, il reprend toutes les méthodes exposées pour en faire des descriptions XML. Bien sûr, chaque méthode native exposée doit correspondre à une méthode de configuration, d'entrée ou de sortie. Enfin, il écrit l'interface publique en XML. Il reprend alors l'interface complémentaire dont il enlève toutes les spécificités de chacune des plates-formes cibles. Dans le chapitre 8, nous présentons des logiciels qui permettent d'automatiser une partie de la phase de développement.

Dans le prochain chapitre, nous allons décrire notre langage commun permettant d'écrire une seule fois le code permettant d'utiliser un composant sur plusieurs plates-formes mobiles. Nous montrerons entres autres comment l'interface commune est utilisée pour arriver à ce résultat.

Chapitre 7

Langage universel et compilateur

Dans notre solution, pour atténuer les différences d'implémentations des composants multiplateformes, les intégrateurs ont à leur disposition un nouveau langage de programmation. Les instructions écrites avec ce langage sont multiplateformes, c'est-à-dire que la même instruction peut être utilisée dans plusieurs environnements de développement (eclipse, Xcode ...) et plusieurs langages de programmation (Java, Objective-C ...). Ainsi, nous cachons les différences entre plates-formes. Dans ce chapitre, nous présentons ce langage et le compilateur associé.

7.1 Introduction

Dans le précédent chapitre, nous avons présenté des composants multiplateformes. Ce nouveau type de composants a la particularité d'avoir une interface qui résume de façon indépendante de toutes les plates-formes mobiles leurs fonctionnalités. Cependant, aujourd'hui, il n'existe pas de langage pour invoquer une méthode à partir de ce type d'interface. Nous présentons donc, dans ce chapitre, un langage commun à plusieurs plates-formes mobiles permettant d'intégrer des composants multiplateformes à partir de leur interface publique. Ce langage commun est indépendant de toutes les plates-formes mobiles. Ainsi, il est possible pour un intégrateur de composants d'écrire une seule fois le code source permettant d'intégrer un composant et ensuite réutiliser ce code source à travers plusieurs applications natives implémentées pour différentes plates-formes. Ce langage est bien sûr accompagné d'un compilateur source à source qui s'occupe de transformer toutes les instructions écrites avec celui-ci en code source natif. Avec ce langage et son compilateur, nous masquons toutes les spécificités des différentes plates-formes mobiles, ce qui est l'objectif final de notre framework de développement.

Le chapitre est organisé en trois parties. La première concerne la description du langage commun avec la section 7.2. La deuxième partie, section 7.3, présente le compilateur associé à ce langage. La troisième partie, section 7.4, concerne, quant à elle, la prise en compte des évolutions du mobile par notre solution. Nous montrons à partir d'un exemple concret que notre solution est suffisamment flexible pour prendre en compte les évolutions futures dans le domaine du développement mobile.

7.2 Langage commun

Dans cette section, nous présentons notre langage commun. Dans un premier temps, nous rappelons les besoins auxquels notre langage doit répondre. Dans un deuxième temps, nous donnons les instructions de base qui composent ce langage ainsi que leurs liens avec les composants multiplateformes présentés dans le chapitre précédent.

7.2.1 Besoins

Dans l'architecture de notre solution, chapitre 5, figure 5.2, la structure des applications utilisant notre framework de développement est réalisée à partir des environnements de développement natifs de chaque plate-forme. Ainsi, les utilisateurs implémentent l'ergonomie (le comportement de l'interface et la définition des vues) de leurs applications et tout ce qui ne peut être fait à partir de nos composants de façon spécifique à chaque plate-forme avec les outils spécifiques de chacune d'elle (eclipse pour Android, XCode pour iOS, ...). Notre langage commun doit donc, en plus de permettre l'intégration de nos composants, être utilisé de façon complémentaire avec des outils natifs sans pour autant modifier ou bouleverser les pratiques de développement instaurées par chacun des environnements.

De plus, notre langage commun doit être suffisamment expressif et flexible pour permettre n'importe quelle sorte d'invocation aux méthodes de configuration et d'entrée d'un composant. Effectivement, il doit être capable d'appeler des méthodes et de récupérer leurs retours si la méthode est une fonction. Il doit aussi être capable de gérer le cas où la méthode n'a pas de paramètre d'entrée ainsi que le cas où elle en a un ou plusieurs.

Ensuite, notre langage commun ne doit pas engendrer de surcoût supplémentaire trop important en matière de temps d'apprentissage. Pour être efficace, nous considérons que l'apprentissage de ce langage et de ses subtilités doit durer moins de deux jours. Sans ça, les entreprises de développement mobile risquent de ne pas adopter notre solution. Le coût de formation de chaque développeur serait trop élevé. À keyneosoft, nous avons déjà refusé d'utiliser certaines solutions pour cette raison.

Pour terminer, les instructions permettant le lancement des fonctionnalités d'un composant doivent être les mêmes sur chacune des plates-formes cibles. C'est ce qui permettra de garantir l'unicité du code source écrit avec notre langage ainsi que de faciliter le développement mobile multiplateforme.

7.2.2 Un langage commun basé sur les Annotations

Avant de commencer la présentation de notre langage commun, nous avions le choix entre deux solutions :

— Soit créer notre langage de programmation.

— Soit surcharger un langage de programmation existant.

La première solution est la plus flexible. En effet, nous pouvons créer notre propre sémantique et syntaxe. Cependant, cette solution obligerait les utilisateurs de notre solution à apprendre un nouveau langage de programmation. Comme nous l'avons souligné précédemment, cet apprentissage ne doit pas dépasser deux jours. Si notre langage de programmation est trop complexe nous risquons de ne pas respecter cette contrainte. Nous avons donc choisi de surcharger un langage de programmation existant. Plus précisément, nous avons surchargé le langage de programmation "Annotation". Bien sûr, nous avons adapté ce langage à nos besoins. Dans la suite, nous expliquons notre choix et les adaptations que nous avons apportés

à l'utilisation de ce langage.

Comme, nous l'avons vu dans le chapitre 4, section 4.4, classiquement, dans la programmation par composants, les annotations servent à ajouter des métadonnées ou à donner un rôle à un objet dans un code source. Nous avions donné l'exemple des EJBs qui utilisent les annotations pour la description des composants [EJB08]. Cependant, dans certains cas, les annotations permettent beaucoup plus de choses. Par exemple, dans la programmation orientée objet en Java, les annotations peuvent influencer le comportement d'une application à l'exécution [GJSB05, GG08]. Aujourd'hui, dans notre proposition, nous n'utilisons pas cet aspect mais nous y reviendrons dans les perspectives.

De plus, les annotations ont la particularité de s'intégrer facilement dans un code source existant. Dans le cas du Java EE, le langage principal est le java et les annotations sont utilisées comme un complément. Cette propriété est très importante pour nous. En effet, nous voulons que notre langage s'intègre dans les environnements de développement de chaque plate-forme cible. Notre langage doit donc être suffisamment flexible pour s'intégrer dans différents langages avec des syntaxes complètement différentes tels que le Java, l'Objective-C et le C#. Heureusement les annotations sont déjà intégrées dans ces langages de programmation.

7.2.3 Les annotations dans le mobile

Les annotations sont déjà intégrées dans les environnements de développement mobiles Android, iOS et Windows Phone 8. Sur Android, et plus particulièrement en Java, elles sont principalement utilisées pour décrire la structure d'un code source. Par exemple, l'annotation ***@override*** permet de déclarer une méthode comme surchargée de la classe mère. De plus, plusieurs projets ayant pour objectif de diminuer le temps de développement sur Android à base d'annotations ont vu le jour. Par exemple, le projet "AndroidAnnotations" [1] permet de simplifier le code source. Ainsi avec ce projet, si un utilisateur veut lancer un service sur un thread secondaire (ce qui équivaut à vingt cinq lignes de code) il peut le faire avec une seule annotation (une seule ligne de code) : ***@Background***. Sur iOS, quant à elles, les annotations sont principalement utilisées pour simplifier le développement d'applications. Par exemple, l'annotation ***@synthesize*** permet de générer automatiquement les getters et setters d'une variable de classe à la compilation sans pour autant surcharger le code source d'une application.

De plus, dans les exemples que nous venons de citer, les annotations sont transformées en code source natif à la compilation. C'est exactement ce processus que nous voulons explorer dans notre framework de développement. L'avantage avec ce processus est qu'il est totalement transparent pour les développeurs. Ils n'ont jamais accès au code source généré et ne peuvent donc pas le modifier.

Parce que les annotations existent déjà dans le monde du développement mobile, en choisissant de les surcharger pour notre langage commun, nous pensons que les développeurs mobiles, c'est-à-dire, les utilisateurs de notre solution, n'auront pas besoin d'un temps important d'apprentissage de notre langage et que cela ne bouleversera pas leurs pratiques habituelles de développement.

1. Site officiel du projet AndroidAnnotations : http ://androidannotations.org

7.2.4 Les annotations dans notre framework de développement

Dans cette section, nous allons présenter les nouvelles annotations que nous avons créées pour intégrer un composant dans une application mobile native.

Invocation d'une fonctionnalité d'un composant

Notre langage doit permettre d'appeler les fonctionnalités de nos composants. Pour cela, nos composants multiplateformes mettent en avant deux types de méthodes : les méthodes de configuration et les méthodes d'entrée. Les méthodes de configuration permettent de paramétrer le composant avant son utilisation avec les méthodes d'entrée. Nous considérons donc que les deux types de méthodes sont liés. Le composant pourra être configuré avant chaque appel à une fonctionnalité. Bien sûr, l'intégrateur peut invoquer une méthode d'entrée sans pour autant configurer le composant. Dans ce cas, les paramètres par défaut du composant seront pris en compte. Cependant, l'inverse n'est pas possible. En effet, nous ne permettons pas qu'une méthode de configuration soit appelée sans qu'une méthode d'entrée soit appelée à la suite. Le composant serait alors configuré mais pas utilisé, ce qui est inutile. En effet, l'application utiliserait plus de ressources que nécessaires. Nous ne traiterons donc pas ce cas.

Par conséquent, nous avons créé une nouvelle annotation nommée ***@method***, voir code 7.1. Cette annotation permet d'appeler une méthode d'entrée d'un composant particulier après l'avoir configuré.

Code 7.1 – Signature de l'annotation ***@method***

```
1 @method(component="componentName",
2         configurations={"configurationMethod1","configurationMethod2", ... "
              configurationMethodN"},
3         method="methodName");
```

L'annotation ***@method*** prend en entrée trois paramètres :

i) Le nom du composant à intégrer. C'est le nom qui est déclaré dans la partie description de l'interface publique du composant à appeler dans la balise *componentName*. Ce paramètre est obligatoire car il est tout à fait possible d'avoir des composants avec les mêmes noms de méthodes de configuration ou d'entrée.

ii) Une liste de méthodes de configuration. Les noms des méthodes fournies ici sont les noms contenus dans la partie *configurations* de l'interface publique du composant dans les balises *method* à l'attribut *methodName*. Il peut y avoir plusieurs méthodes de configuration appelées. Dans ce cas, notre solution va traiter chaque méthode de configuration dans l'ordre dans lequel elles ont été déclarées. Ce paramètre est non obligatoire.

iii) La méthode d'entrée correspondant à la fonctionnalité à invoquer. Le nom de la méthode fournie ici équivaut au nom contenu dans la partie *methods* de l'interface publique du composant dans la balise *method* à l'attribut *methodName*. Ce paramètre est obligatoire.

L'annotation ***@method*** se place devant la déclaration d'une variable lorsque la méthode d'entrée appelée est une fonction et retourne donc une valeur. Dans ce cas, le résultat de la fonction sera transmis à la variable annotée. Nous reviendrons sur ce point lors de la présentation de notre compilateur. Si la méthode d'entrée appelée n'est pas une fonction, l'annotation ***@method*** se place n'importe où dans un code source sans aucune restriction. Cette déclaration est complètement indépendante d'une plate-forme cible.

Avec l'annotation ***@method***, les intégrateurs de nos composants peuvent déclarer une méthode. Cependant, ils ne peuvent pas fournir les paramètres d'entrée des méthodes de

configuration et d'entrée du composant. Pour ce faire, nous avons créé une nouvelle annotation nommée ***@var***.

Code 7.2 – Signature de l'annotation ***@var***

```
1 @var(component="componentName",
2        method="methodName",
3        parameter="parameterName");
```

L'annotation ***@var*** prend en entrée trois paramètres obligatoires :

i) Le nom du composant à intégrer. C'est le nom qui est déclaré dans la partie description de l'interface publique du composant à appeler dans la balise *componentName*. Ce paramètre est obligatoire car il est tout à fait possible d'avoir des composants avec les mêmes noms de méthodes et de paramètres d'entrée.

ii) Le nom de la méthode de configuration ou d'entrée dans laquelle le paramètre sera utilisé. Ce paramètre est obligatoire. Sans lui, il ne pourrait pas être associé à une méthode. Les noms des méthodes fournies ici sont les noms contenus dans la partie *configurations* ou *methods* de l'interface publique du composant dans la balise *method* à l'attribut *methodName*.

iii) Le nom du paramètre d'entrée associé à la variable. Ce nom de paramètre se trouve dans l'interface publique du composant dans les balises *parameter* à l'attribut *parameterName* des méthodes concernées.

Cette annotation se place devant une variable déclarée nativement dans un code source. Elle permet de lier cette variable à un paramètre d'entrée d'une méthode. Cette déclaration est donc dépendante d'une plate-forme cible. En effet, la variable qui sera annotée avec l'annotation ***@var*** devra avoir le même type que le paramètre d'entrée auquel elle sera liée. Si ce n'est pas le cas, notre compilateur retournera une erreur. Pour atténuer les éventuelles différences entre types d'objets dans les différents systèmes d'exploitation, nous permettons uniquement la déclaration des paramètres fonctionnels d'une méthode, c'est-à-dire, les paramètres définis dans l'interface publique du composant. Tous les paramètres non fonctionnels seront déduits en fonction du contexte de compilation. Bien sûr, la même variable déclarée dans un code source pourra être annotée par plusieurs annotations ***@var*** qui représenteront chacune une méthode de configuration ou d'entrée différente.

L'annotation ***@var*** traite les paramètres d'entrée "classiques". Les paramètres de type delegate sont, quant à eux, traités avec une autre annotation.

Gestion de la délégation

Dans notre solution, nos composants peuvent déléguer certaines parties de leurs processus. Dans le chapitre précédent, nous nous sommes limités à la délégation des retours. Un composant comme le "httpRequestManager" délègue, par exemple, le traitement de la réponse finale qu'il a reçu des différents web services appelés. Le système de délégation dans notre solution n'est pas uniquement limité à la récupération des retours du composant. En effet, dans certains cas, nos composants ne peuvent pas effectuer une partie d'un processus car cette partie est trop spécifique à une application. Si le composant traiterait cette partie du processus, il ne pourrait être utilisé que dans une seule application.

En matière de programmation, nous avons vu dans le chapitre précédent qu'un objet de type delegate est représenté par une liste de méthodes. Cette liste de méthode doit être implémentée par l'application utilisant le composant. Sur Android, cela est représenté par une

interface qu'une classe de l'application doit implémenter. Sur iOS, cela est représenté par un protocole que l'application hôte doit implémenter. En matière d'annotation, nous permettons à l'intégrateur d'annoter une classe avec l'annotation ***@delegate***.

Code 7.3 – Signature de l'annotation ***@delegate***

```
1 @delegate(component="componentName",
2           method="methodName",
3           parameter="parameterName");
```

Cette annotation prend en entrée trois paramètres obligatoires :

i) Le nom du composant à intégrer. C'est le nom qui est déclaré dans la partie description de l'interface publique du composant à appeler dans la balise *componentName*. Ce paramètre est obligatoire.

ii) Le nom de la méthode d'entrée qui devra utiliser cette classe comme delegate. Ce paramètre est obligatoire, sans ça nous ne pouvons pas affecter le traitement du delegate à une méthode d'entrée. Les noms des méthodes fournies ici sont les noms contenus dans la partie *methods* de l'interface publique du composant dans la balise *method* à l'attribut *methodName*.

iii) Le nom du paramètre d'entrée dans la méthode d'entrée souhaitée. Ce paramètre est obligatoire. Ce nom de paramètre se trouve dans l'interface publique du composant dans les balises *parameter* à l'attribut *parameterName* des méthodes concernées. Il doit correspondre à un delegate sinon notre compilateur retournera une erreur.

L'annotation ***@delegate*** peut être déclarée devant deux types d'instructions dans un code source natif : soit elle est placée devant une variable, soit elle est placée devant une classe. Dans le premier cas, l'annotation fournit le même comportement qu'une annotation ***@var***. Elle permettra de lier la variable annotée avec un paramètre d'entrée. Dans le deuxième cas, elle permettra de lier l'objet courant au paramètre d'entrée.

Pour récapituler, l'intégrateur souhaitant appeler une fonctionnalité d'un de nos composants multiplateformes devra utiliser des annotations dans son code source qu'il soit sur Android, iOS ... Dans un premier temps, il devra appeler une méthode d'entrée du composant avec l'annotation ***@method***. Il choisira alors, les méthodes de configuration qu'il souhaite utiliser pour paramétrer le composant. Ensuite, avec l'annotation ***@var***, il devra déclarer tous les paramètres d'entrée de la méthode d'entrée et des éventuelles méthodes de configuration à intégrer. Il devra faire un traitement à part pour les paramètres d'entrée de type delegate. En effet, il devra déclarer les classes qui implémentent les délégates avec l'annotation ***@delegate***.

Liens entre notre langage commun et l'interface publique d'un composant

L'avantage de notre solution est d'écrire une seule fois le code source pour intégrer nos composants et ensuite être capable de le réutiliser sur plusieurs environnements de développement différents. Pour ce faire, nous nous basons sur l'interface publique de nos composants qui est indépendante de n'importe quelle plate-forme cible. Ainsi sur la figure 7.1, nous récapitulons tous les liens entre nos annotations et les interfaces publiques de nos composants.

Les liens entre l'interface publique et notre langage nous permettent d'intégrer facilement n'importe quel composant. C'est l'objet de la prochaine section. En effet, nous allons reprendre le composant multiplateforme "HttpRequestManager" pour l'intégrer dans le code source d'une application Android et d'une application iOS.

Exemple d'interface publique

```
<?xml version="1.0" encoding="UTF-8"?>
<Component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="CommonInterfaceSchema.xsd" >
   <description>
      <componentName>componentName</componentName>
   </description>

   <configurations>
      <method name="configurationMethodName" description="...">
         <parameter identifier="identifier1"
                    io="in"
                    description="..."/>
      </method>
   </configurations>
   <methods>
      <method name="methodName" description="...">
         <parameter identifier="identifier2"
                    io="in"
                    description="..."/>
         <parameter identifier="identifier3"
                    io="in"
                    description="..."
                    delegate="true"/>
      </method>
   </methods>
</Component>
```

Annotation *@method* possible

```
@method(component="componentName", configurations={"configurationMethodName"},
           method="methodName");
```

Annotation *@var* et *@delegate* nécessaire à la méthode appelée

```
@var(component="componentName", method="methodName", parameter="identifier1");

@var(component="componentName", method="methodName", parameter="identifier2");

@delegate(component="componentName", method="methodName",
           parameter="identifier3");
```

FIGURE 7.1 – Liens entre l'interface publique d'un composant et nos annotations

7.2.5 Intégration sous Android et iOS du composant HttpRequestManager avec notre langage commun

Dans cette section, nous allons nous focaliser sur l'intégration de la méthode d'entrée ***sendHttpRequest(request, delegate)*** du composant car cette méthode permet d'illustrer toutes nos annotations. En effet, la méthode a des paramètres d'entrée dont un delegate ce qui nous permet d'utiliser les annotations ***@var*** et ***@delegate***. De plus pour appeler la méthode nous utiliserons l'annotation ***@method***.

Définition des annotations à intégrer dans un code source

Tout d'abord, l'intégration de la méthode d'entrée se fait avec l'annotation ***@method*** comme présentée ci-dessous, code 7.4 :

Code 7.4 – Appel de la méthode ***sendHttpRequest*** avec l'annotation ***@method***

```
1 @method(component="HttpRequestManager", method="sendHttpRequest");
```

Après avoir défini la méthode d'entrée qui doit être appelée, il faut définir les paramètres d'entrée de la méthode. Pour cela, il faut utiliser l'annotation ***@var*** ainsi que l'annotation ***@delegate***. En effet, le deuxième paramètre d'entrée de la méthode ***sendHttpRequest(request,delegate)*** est de type delegate. Cette information se trouve dans la deuxième balise *parameter* de la méthode dans l'interface publique du composant. Voici les deux annotations à intégrer dans un code source natif.

Code 7.5 – Annotations permettant de finaliser l'appel au composant "httpRequestManager"

```
1 @var(component="HttpRequestManager", method="sendHttpRequest", parameter="
      request");
2
3 @delegate(component="HttpRequestManager", method="sendHttpRequest",
      parameter="delegate");
```

Toutes les informations entrées lors de l'appel de cette méthode sont récupérées à partir de l'interface publique du composant dans les balises *description* et *methods*. Sur la figure 7.2, nous faisons le parallèle entre l'interface publique du composant et les annotations permettant de l'intégrer dans une application native.

Interface publique

```
<?xml version="1.0" encoding="UTF-8"?>
<Component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="CommonInterfaceSchema.xsd" >

           <description>
                      <componentName>HttpRequestManager</componentName>
           </description>

           <methods>
                      <method name="sendHttpRequest" description="...">
                                 <parameter identifier="request"
                                            io="in"
                                            description="request to send"/>
                                 <parameter identifier="delegate"
                                            io="in"
                                            description="method delegate"
                                            delegate="true"/>
                      </method>
           </methods>
</Component>
```

Annotation *@method*

```
@method(component="HttpRequestManager", method="sendHttpRequest");
```

Annotation *@var* et *@delegate*

```
@var(component="HttpRequestManager", method="sendHttpRequest",
           parameter="request");

@delegate(component="HttpRequestManager", method="sendHttpRequest",
           parameter="delegate");
```

FIGURE 7.2 – Lien entre l'interface publique du composant HttpRequestManager et les annotations permettant de l'intégrer dans une application native

Intégration dans un code source natif.

Le composant "HttpRequestManager" peut être intégré dans un code source natif Android et iOS. Nous allons donc montrer dans cette section où intégrer les annotations que nous avons définies précédemment.

Tout d'abord, définissons le contexte de l'intégration. Nous avons implémenté une application ayant une seule vue qui s'occupe d'envoyer une requête HTTP de type get à un web service lorsque l'on clique sur un bouton. Dans chacune des versions de l'application, nous avons un contrôleur de vue qui s'occupe de récupérer les interactions utilisateurs. Ici, nous nous sommes limités à la récupération des clics de l'utilisateur. À chaque récupération de l'évènement, nous appelons un web service. L'application contient aussi un fichier par vue (un fichier XML pour Android, un fichier storyboard pour iOS). Pour cette intégration, nous allons nous limiter au code du contrôleur de vue présentées sur les codes 7.6 et 7.7.

Code 7.6 – Code du contrôleur de vue dans notre application d'intégration Android

```java
package com.keyneosoft.httprequestmanagerintegration;

import android.os.Bundle;
import android.support.v7.app.ActionBarActivity;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button launchRequestButton = (Button)this.findViewById(R.id.
            launchRequestButton);
        launchRequestButton.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View v) {
                // get users' clicks
            }
        });
    }
}
```

Code 7.7 – Code du contrôleur de vue dans notre application d'intégration iOS

```objc
#import "ViewController.h"

@interface ViewController ()

@end

@implementation ViewController

-(IBAction)onButtonClick:(id)sender {
    // get users' clicks
}

- (void)viewDidLoad
{
    [super viewDidLoad];
```

```
16 }
17
18 - (void)didReceiveMemoryWarning
19 {
20     [super didReceiveMemoryWarning];
21 }
22
23 @end
```

Comme il est montré dans les codes 7.6 et 7.7, la récupération des clics utilisateurs se fait sur Android et iOS respectivement à l'intérieur des méthodes ***onClick(View)*** et ***onButtonClick :***. C'est dans ces deux méthodes qu'il faudra appeler l'annotation définie dans le code 7.4. Avant d'intégrer la méthode, il faut aussi définir les paramètres d'entrée qui sont une requête HTTP et un delegate qui sont respectivement sur Android et iOS de type HttpGet et NSURLRequest. Les informations sur les types des variables se trouvent dans l'interface complémentaire du composant, chapitre 6, code 6.5. Pour autant, l'intégrateur n'a pas besoin d'aller récupérer cette information dans cette interface. En effet, nous verrons, dans le chapitre 8, que c'est un logiciel qui est chargé de retranscrire toutes les informations nécessaires à l'intégration.

Dans notre exemple, le delegate sera la classe courante, c'est-à-dire, notre contrôleur de vue. L'annotation ***@delegate*** est donc placée devant la déclaration de la classe. Le code source final est présenté avec les codes 7.8 et 7.9.

Code 7.8 – Code du contrôleur de vue dans notre application d'intégration Android

```
1  package com.keyneosoft.httprequestmanagerintegration;
2
3  import org.apache.http.client.methods.HttpGet;
4
5  import android.os.Bundle;
6  import android.support.v7.app.ActionBarActivity;
7  import android.view.View;
8  import android.view.View.OnClickListener;
9  import android.widget.Button;
10
11 @delegate(component="httpRequestManager", method="sendHttpRequest",
       parameter="delegate")
12 public class MainActivity extends ActionBarActivity
13 {
14
15     @Override
16     protected void onCreate(Bundle savedInstanceState) {
17         super.onCreate(savedInstanceState);
18         setContentView(R.layout.activity_main);
19
20         Button launchRequestButton = (Button)this.findViewById(R.id.
               launchRequestButton);
21         launchRequestButton.setOnClickListener(new OnClickListener() {
22
23             @Override
24             public void onClick(View v) {
25                 // get users' clicks
26                 @var(component="httpRequestManager" , method="sendHttpRequest"
                       , parameter=" request")
27                 HttpGet getRequest = new HttpGet("url");
28
29                 @method(component="httpRequestManager", method="sendHttpRequest
                       ")
30             }
31         });
```

```
32     }
33 }
```

Code 7.9 – Intégration d'un appel au "HttpRequestManager" dans un code source natif iOS

```
1 #import "ViewController.h"
2
3 @delegate(component="httpRequestManager", method="sendHttpRequest",
      parameter="delegate")
4 @interface ViewController ()
5
6 @end
7
8 @implementation ViewController
9
10 -(IBAction)onButtonClick:(id)sender {
11     // get users' clicks
12     @var(component="httpRequestManager" , method="sendHttpRequest" ,
          parameter=" request")
13     NSURLRequest *getRequest = [[NSURLRequest alloc] initWithURL:[NSURL
          URLWithString:@"url"]];
14
15     @method(component="httpRequestManager", method="sendHttpRequest")
16 }
17
18 - (void)viewDidLoad
19 {
20     [super viewDidLoad];
21 }
22
23 - (void)didReceiveMemoryWarning
24 {
25     [super didReceiveMemoryWarning];
26 }
27
28 @end
```

Dans cette section, nous avons présenté les annotations permettant d'intégrer les fonctionnalités de nos composants. Elles permettent d'intégrer une méthode d'entrée, de lui affecter des méthodes de configuration et de sortie via un système de délégation. Les annotations générées ont uniquement besoin de l'interface publique de nos composants pour être remplies, ce qui nous permet de fournir un langage commun et indépendant à plusieurs plates-formes. Dans la suite, nous présentons notre compilateur qui s'occupe notamment d'interpréter et transformer le code source écrit avec nos annotations en langage natif.

7.3 Un compilateur source à source flexible et léger

Nous voulons que notre solution fournisse des applications 100% natives. Pour arriver à ce résultat, nous devons fournir un compilateur source à source qui s'occupera de générer le code source natif correspondant au code source écrit avec nos annotations. Dans cette section, nous allons tout d'abord résumer les besoins auxquels le compilateur doit répondre. Ensuite, nous aborderons les règles de compilation qu'il doit implémenter.

7.3.1 Besoins

Nous souhaitons que notre framework de développement fournisse uniquement des applications qui ne contiennent que du code natif, chapitre 5, section 5.1. En effet, nous ne voulons pas que notre solution apporte un surcoût au temps d'exécution des applications générées avec elle à cause d'une exécution à la volée de certaines parties du code pour plusieurs plates-formes cibles. Pour ce faire, il nous faut un compilateur source à source qui s'occupe de transformer le code écrit avec notre langage commun en code natif spécifique à chaque plate-forme cible. Ce compilateur doit être multiplateforme. Effectivement, il devra fonctionner sur Windows 8 pour compiler les applications Windows Phone 8 utilisant notre solution ainsi que sur mac pour les applications iOS.

De plus, le compilateur doit fonctionner en mode pré-compilation, c'est-à-dire, qu'il fonctionnera avant le compilateur officiel de chaque plate-forme cible. Il ne s'occupera donc que de la transformation de notre langage commun en code source natif et non de la transformation du code source natif complet en exécutable pouvant être exécuté sur des smartphones. Ce type de processus est très utilisé dans les solutions existantes telles que PhoneGap, Titanium mobile ...

Enfin, ce compilateur doit être basé sur des règles de transformations simples à mettre en oeuvre. Si une nouvelle plate-forme ou une nouvelle version d'un système d'exploitation mobile existant voit le jour, nous devons être capables très rapidement de les prendre en compte dans notre compilateur. Pour être accepté dans une entreprise mobile, nous estimons à une semaine maximum le temps acceptable de mise à jour de notre compilateur pour la sortie d'une nouvelle version de système d'exploitation et un mois pour la prise en compte d'une nouvelle plate-forme mobile ou d'un nouveau langage de programmation.

7.3.2 Règles de compilation

Parmi les trois annotations ***@method***, ***@var*** et ***@delegate***, seule l'annotation ***@method*** doit être transformée en langage natif. En effet, c'est cette annotation qui permet d'appeler une fonctionnalité du composant. Les autres permettent uniquement de lier une variable ou une classe à un paramètre d'entrée d'une méthode d'entrée ou de configuration.

Transformation d'une annotation *@method* en code natif sur plusieurs plates-formes

Pour transformer une annotation de type ***@method***, plusieurs étapes sont nécessaires :

i) Instancier le composant ou plutôt la classe qui implémente la méthode appelée. Dans ce cas, il existe deux possibilités. Si la classe est présentée comme un singleton, le compilateur récupère l'instance globale de la classe sinon il crée une nouvelle instance grâce aux méthodes d'instanciation native.

ii) Configurer l'objet instancié avec les méthodes de configuration renseignées dans l'annotation ***@method*** dans le champ *configurations*.

iii) Appeler la méthode renseignée dans l'annotation.

iv) Si la méthode appelée est une fonction, le compilateur associe l'appel de la méthode à la variable annotée par l'annotation ***@method***.

Bien sûr, les paramètres d'entrée des méthodes d'entrée et de configuration sont remplis grâce aux objets natifs annotés avec les annotations ***@var*** et ***@delegate***. Pour générer le code

natif, le compilateur se base uniquement sur l'interface complémentaire. En effet, c'est dans celle-ci que toutes les informations sur les implantations natives du composant se trouvent.

Nous illustrons d'ailleurs la transformation d'annotations en code natif avec la figure 7.3. Nous avons représenté chaque étape de la transformation. Pour cela, nous sommes partis de l'interface complémentaire d'un composant et des annotations pour intégrer une de ses méthodes d'entrée jusqu'au code source généré sur Android et iOS. Bien sûr, nous nous sommes limités aux éléments utiles de l'interface complémentaire pour notre exemple. En jaune et vert, nous avons surligné le nom des méthodes d'entrée et de configuration qui doivent être intégrées. Nous remarquons rapidement que pour chaque implémentation le nom de la méthode native peut être différent. C'est cet élément qui est surligné en bleu. En rouge, nous avons surligné tous les éléments liés aux types natifs de la plate-forme cible. Ce sont ces types auxquels doivent correspondre les paramètres d'entrée d'une méthode d'entrée ou de configuration. Enfin, en orange, nous avons mis en évidence les liens entre les paramètres d'entrée d'une méthode et les annotations ***@var***.

Gestion de la délégation

Dans notre solution, un delegate est représenté sous la forme d'une liste de méthodes que l'intégrateur de composant doit implémenter. Nous avons vu dans la section précédente qu'il peut annoter une classe avec l'annotation ***@delegate***. C'est avec cette annotation que notre compilateur va générer pour l'intégrateur les instructions nécessaires pour que la classe annotée implémente les méthodes du délégate.

Sur Android, nous avons vu qu'un délégate est représenté dans une interface java. Sur iOS, il est représenté par un protocole. Le compilateur sera donc chargé de générer tout le code pour implémenter l'interface ou le protocole en question. La gestion des méthodes du délégate se fera ensuite de façon native par l'intégrateur. Nous donnons un exemple de ce traitement dans la section 7.3.3.

Gestion contextuelle de la compilation

Dans le chapitre 6 à la section 6.5, nous avons différencié deux types de paramètres : les paramètres fonctionnels et les paramètres non fonctionnels. Les paramètres fonctionnels d'une méthode sont les paramètres qui servent pendant le traitement de la méthode. Par exemple, la méthode ***sendHttpRequest(request, delegate)*** du composant "HttpRequestManager" a obligatoirement besoin en entrée d'une requête HTTP. Sans ça elle ne pourra pas effectuer le traitement souhaité. Quant à eux, les paramètres non fonctionnels sont tous les paramètres qui n'apportent rien au traitement mais qui sont cependant obligatoires pour que le traitement fonctionne sur la plate-forme cible.

Par exemple, la version de la méthode ***sendHttpRequest(request, delegate)*** sur Android est implémentée de deux façons natives différentes ***sendHttpGetRequest(Context, HttpGet, HttpGetRequestServiceDelegate)*** et ***sendHttpPostRequest(Context, HttpPost, HttpPostRequestServiceDelegate)***. Ces deux méthodes attendent un objet de type Context en entrée. Cet objet permet de lancer la requête mais n'est pas pour autant primordial pour l'utilisateur du composant. De plus, cet objet n'existe même pas sur les autres plates-formes mobiles. Nous devons donc le cacher aux intégrateurs de nos composants. De la même façon, l'intégrateur ne doit pas savoir que la méthode ***sendHttpRequest(request, delegate)*** est implémentée de deux façons différentes. Tout cela doit être transparent.

Android (JAVA)

Interface complémentaire

```
<extraInformationOnConfigurationMethod
   name="genericConfigurationMethodName">
   <android name="androidConfigurationMethodName"
            class="ClassName"
            package="com.company" >
   </android>
</extraInformationOnConfigurationMethod>

<extraInformationOnMethod
   name="genericMethodName">
   <android name="androidMethodName"
            class="ClassName"
            package="com.company"
            isSingleton="false" >
      <parameter identifier="identifier1"
                 type="Type1"
                 io="in"
                 functional="true"
                 delegate="false"/>
      <parameter identifier="identifier2"
                 type="Type2"
                 io="in"
                 functional="true"
                 delegate="false"/>
   </android>
</extraInformationOnMethod>
```

iOS (Objective-C)

Interface complémentaire

```
<extraInformationOnConfigurationMethod
   name="genericConfigurationMethodName">
   <iOS name="iOSConfigurationMethodName"
        class="ClassName" >
   </iOS>
</extraInformationOnConfigurationMethod>

<extraInformationOnMethod
   name="genericMethodName">
   <iOS name="iOSMethodName:arg2:"
        class="ClassName"
        isSingleton="false" >
      <parameter identifier="identifier1"
                 type="Type1"
                 io="in"
                 functional="true"
                 delegate="false"
                 argument="iOSMethodName"/>
      <parameter identifier="identifier2"
                 type="Type2"
                 io="in"
                 functional="true"
                 delegate="false"
                 argument="arg2"/>
   </iOS>
</extraInformationOnMethod>
```

Annotations nécessaires à l'appel de la méthode *genericMethodName*

```
@var(component="componentName", method="genericMethodName", parameter="identifier1 »)
@var(component="componentName", method="genericMethodName", parameter="identifier2 »)
@method(component="componentName", configurations={"genericConfigurationMethodName"},
        method="genericMethodName")
```

Dans un code source Android

```
@var(component="componentName",
     method="genericMethodName",
     parameter="identifier1")
Type1 param1 = new Type1(...);

@var(component="componentName",
     method="genericMethodName",
     parameter="identifier2")
Type2 param2 = new Type2(...);

@method(component="componentName",
     configurations={"genericConfigurationMethodName"},
     method="genericMethodName")
```

Dans un code source iOS

```
@var(component="componentName",
     method="genericMethodName",
     parameter="identifier1")
Type1 param1 = [[Type1 alloc] init];

@var(component="componentName",
     method="genericMethodName",
     parameter="identifier2")
Type2 param2 = [[Type2 alloc] init];

@method(component="componentName",
     configurations={"genericConfigurationMethodName"},
     method="genericMethodName")
```

Code généré sur Android

```
@method(component="componentName",
   configurations={"genericConfigurationMethodName"},
   method="genericMethodName")
ClassName component = new ClassName();
component1.androidConfigurationMethodName();
component1.androidMethodName(param1, param2);
```

Code généré sur iOS

```
@method(component="componentName",
     configurations={"genericConfigurationMethodName"},
     method="genericMethodName")
ClassName component = [[ClassName alloc] init];
[component1 iOSConfigurationMethodName];
[component1 iOSMethodNameArg1:param1
            arg2:param2];
```

FIGURE 7.3 – Liens entre l'interface complémentaire d'un composant et le code généré par le compilateur

Pour cacher les différentes implémentations d'une même méthode ou les paramètres non fonctionnels, le compilateur doit déduire et choisir automatiquement les bons champs à générer. Pour cela, le compilateur a à sa disposition un contexte de complication qui contient plusieurs types d'informations :

— La plate-forme pour laquelle l'application est compilée.
— Les types des paramètres d'entrée des méthodes appelées.
— L'endroit où la méthode est appelée (le package, la classe, la méthode ...).

— Le type d'exécutable généré (application mobile, libraire Android, framework ou librairie statique iOS ...).

Toutes ces informations vont permettre au compilateur de déduire les bonnes valeurs à remplacer pour les paramètres non fonctionnels. Si nous reprenons l'exemple du contexte dans la méthode ***sendHttpRequest(Context, ...)*** sur Android, si nous sommes dans le contexte suivant :

— Plate-forme cible : android.
— Type du paramètre à déduire : Context.
— Appel dans une classe de type Activity.
— Exécutable généré : une application mobile.

Le compilateur devra remplacer le paramètre d'entrée de type **Context** par ***this.getApplicationContext()***. Si nous prenons un autre contexte de compilation :

— Plate-forme cible : android ;
— Type du paramètre à déduire : Context ;
— Appel dans une classe de type Fragment ;
— Exécutable généré : une application mobile.

Dans ce cas, le compilateur devra remplacer le paramètre d'entrée de type **Context** par ***this.getActivity().getApplicationContext()***.

De la même façon, le compilateur pourra choisir la méthode à appeler entre ***sendHttpGetRequest(Context, HttpGet, HttpGetRequestServiceDelegate)*** et ***sendHttpPostRequest(Context, HttpPost, HttpPostRequestServiceDelegate)*** en fonction du paramètre d'entrée annoté par l'intégrateur. Si le paramètre annoté est de type **HttpGet**, il choisira la première méthode pour la génération de la méthode, sinon la deuxième. Si le paramètre annoté est différent de **HttpGet** ou **HttpPost**, le compilateur remontera une erreur de compilation.

Grâce au contexte de compilation, notre compilateur est capable de gérer de façon contextuelle et automatique tous les objets et méthodes ayant des spécificités à une plate-forme cible. Ainsi, nous nous assurons que l'intégration d'un composant se fera de la même façon sur n'importe quelle plate-forme.

7.3.3 Génération du code source pour l'appel au composant "HttpRequestManager"

Prenons l'exemple du composant "HttpRequestManager", dans la section 7.2.5, nous avons intégré les annotations nécessaires à l'appel de la méthode ***sendHttpRequest(request, delegate)*** du composant. Par rapport à l'exemple donné sur la figure 7.3, le composant "HttpRequestManager" est présenté comme un singleton et fournit les retours de son processus à travers un delegate. Par conséquent, au lieu de créer une instance du composant, le compilateur va rechercher l'instance globale du composant ainsi que générer les méthodes du delegate dans la classe qui l'implémente. La classe courante sera utilisée comme paramètre d'entrée de la méthode ***sendHttpRequest(request, delegate)***. Finalement, le code source final après l'utilisation de notre compilateur est présenté avec les codes 7.10 et 7.11. Bien sûr, le compilateur lie aussi l'exécutable du composant pour chaque plate-forme cible au code source de l'application.

Code 7.10 – Code généré après l'intégration du composant "HttpRequestManager" (Android)

```
1 package com.keyneosoft.httprequestmanagerintegration;
2
3 import org.apache.http.HttpResponse;
```

```
4  import org.apache.http.client.methods.HttpGet;
5
6  import android.app.Activity;
7  import android.os.Bundle;
8  import android.view.View;
9  import android.view.View.OnClickListener;
10 import android.widget.Button;
11
12 import com.keyneosoft.requestManager.delegates.
       HttpGetRequestServiceDelegate;
13 import com.keyneosoft.requestManager.implementations.HttpRequestService;
14
15 public class MainActivity extends Activity implements
       HttpGetRequestServiceDelegate
16 {
17     @Override
18     protected void onCreate(Bundle savedInstanceState) {
19         super.onCreate(savedInstanceState);
20         setContentView(R.layout.activity_main);
21
22         Button launchRequestButton = (Button)this.findViewById(R.id.
               launchRequestButton);
23         launchRequestButton.setOnClickListener(new OnClickListener() {
24
25             @Override
26             public void onClick(View v) {
27                 // get users' clicks
28                 HttpGet getRequest = new HttpGet("url");
29
30                 HttpRequestService requestManager = HttpRequestService.
                       getSharedInstance();
31                 requestManager.sendHttpGetRequest(getApplicationContext(),
                       getRequest, MainActivity.this);
32             }
33         });
34     }
35
36     @Override
37     public void textRecieved(String text, HttpResponse response, HttpGet
           request) {
38         // delegate method
39     }
40
41     @Override
42     public void requestCancelled(HttpGet request) {
43         // delegate method
44     }
45
46     @Override
47     public void requestError(Exception error, HttpGet request) {
48         // delegate method
49     }
50
51     @Override
52     public void statusError(HttpResponse response, HttpGet request) {
53         // delegate method
54     }
55
56     @Override
57     public void noActiveNetworkConnection(HttpGet request) {
58         // delegate method
59     }
60 }
```

Code 7.11 – Code généré après l'intégration du composant "HttpRequestManager" (iOS)

```
#import "ViewController.h"

#import "HttpRequest.h"

@interface ViewController ()

@end

@implementation ViewController

-(IBAction)onButtonClick:(id)sender {
    // get users' clicks
    NSURLRequest *getRequest = [[NSURLRequest alloc] initWithURL:[NSURL
        URLWithString:@"url"]];

    HttpRequest *requestManager = [HttpRequest sharedInstance];
    [requestManager sendHttpRequest:getRequest delegate:self];
}

-(void)textReceived:(NSString*)text
         responding:(NSHTTPURLResponse*)response
                 to:(NSURLRequest*)request {
    // delegate method
}

-(void)requestCancelled:(NSURLRequest*)request {
    // delegate method
}

-(void)requestError:(NSError*)error
                 to:(NSURLRequest*)request {
    // delegate method
}

-(void)statusError:(NSHTTPURLResponse*)response
                to:(NSURLRequest*)request {
    // delegate method
}

-(void)noActiveNetworkConnection:(NSURLRequest*)request {
    // delegate method
}

- (void)viewDidLoad
{
    [super viewDidLoad];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
}

@end
```

7.4 Un framework de développement facilement adaptable

Maintenant que tous les éléments de notre framework sont présentés, prenons un peu de recul sur la solution proposée. Dans les besoins que nous avons spécifiés dans le chapitre 2 section 2.5, nous voulons une solution facilement adaptable et par conséquent capable de prendre en compte les nouveautés du domaine du mobile. Les nouveautés dans le développement mobile se situent à deux niveaux :

— Une nouvelle plate-forme mobile peut voir le jour avec un nouvel environnement de développement associé.
— Un environnement de développement actuel peut évoluer. Souvent, ces types d'évolutions sont mineurs mais il se peut, par exemple, qu'un nouveau langage de programmation voit le jour dans un environnement de développement déjà éprouvé.

Pour prendre en compte ces deux cas, nous avons spécifié un nouveau langage basé sur les annotations. Ces annotations peuvent être intégrées n'importe où dans un code source natif et surtout dans n'importe quel langage de programmation. Si une nouvelle plate-forme voit le jour où un environnement de développement actuel évolue, nous serons toujours capables d'y intégrer notre langage commun. Les annotations sont suffisamment flexibles pour cela.

De plus, le processus de transformation d'une annotation en langage natif est volontairement simple à mettre en oeuvre. Effectivement, les différentes étapes et règles de compilation présentées précédemment sont complètement basées sur l'interface complémentaire des composants. Le compilateur interprète les annotations ***@method*** trouvées dans le code source natif et vérifie qu'elles sont correctes par rapport aux informations présentes dans l'interface complémentaire du composant. Si c'est le cas, le compilateur va, pour chaque paramètre d'entrée de la méthode, vérifier que celui-ci est déclaré avec une annotation ***@var*** ou ***@delegate***. Si c'est le cas, il transforme l'annotation ***@method*** en code natif. Le compilateur doit uniquement retranscrire les informations trouvées dans l'interface complémentaire. C'est un processus léger et facile à mettre en place.

Prenons un exemple concret d'évolution soudaine du domaine mobile. En juin 2014, Apple pendant la keynote de WWDC'14 a présenté un nouveau langage de programmation pour iOS : Swift. Même si aujourd'hui, Apple ne le dit pas clairement, ce nouveau langage risque de prendre la place de l'Objective-C. Notre solution doit donc le prendre en compte le plus tôt possible. Notre langage de programmation basé sur les annotations peut facilement s'intégrer dans ce langage. L'implémentation des composants iOS ne changent pas non plus,. Le swift est compatible avec l'objective-C. Le seul changement se trouve donc au niveau de la transformation des annotations en langage natif ici le Swift. Au lieu de retranscrire l'interface complémentaire en Objective-C, le compilateur doit le faire en Swift. Sur la figure 7.4, nous reprenons l'exemple donné sur la figure 7.3 avec la transformation de l'annotation ***@method*** en Swift. Les informations nécessaires à cette transformation se trouvent toujours dans l'interface complémentaire. Seul le code généré change de syntaxe. Il s'agit donc uniquement d'un changement de syntaxe. Ce changement de syntaxe est d'ailleurs très léger à intégrer dans notre compilateur. Pour récapituler, une évolution majeure d'Apple qui est l'apparition d'un nouveau langage de programmation pour iOS engendre très peu de changements dans notre solution et surtout nous ne changeons pas son architecture.

Bien sûr, si une nouvelle plate-forme voit le jour, la prise en compte sera plus profonde. Plusieurs étapes seront nécessaires :

— Ré-implémenter les composants existants pour cette nouvelle plate-forme.
— Mettre à jour le modèle de l'interface complémentaire si besoin. Il peut y avoir des différences entre les définitions des méthodes pour les différentes plates-formes. C'est

iOS (Swift)

Interface complémentaire

```
<extraInformationOnConfigurationMethod
   name="genericConfigurationMethodName">
   <iOS name="iOSConfigurationMethodName" class="ClassName" >
   </iOS>
</extraInformationOnConfigurationMethod>

<extraInformationOnMethod
   name="genericMethodName">
   <iOS name="iOSMethodName:arg2:" class="ClassName" isSingleton="false" >
      <parameter identifier="identifier1"
                 type="Type1"
                 io="in"
                 functional="true"
                 delegate="false"
                 argument="iOSMethodName"/>
      <parameter identifier="identifier2"
                 type="Type2"
                 io="in"
                 functional="true"
                 delegate="false"
                 argument="arg2"/>
   </iOS>
</extraInformationOnMethod>
```

Annotations nécessaires à l'appel de la méthode *genericMethodName*

```
@var(component="componentName", method="genericMethodName", parameter="identifier1 »)
@var(component="componentName", method="genericMethodName", parameter="identifier2 »)
@method(component="componentName", configurations={"genericConfigurationMethodName"},
        method="genericMethodName")
```

Dans un code source iOS (Swift)

```
@var(component="componentName", method="genericMethodName", parameter="identifier1")
var param1 = Type1()

@var(component="componentName", method="genericMethodName", parameter="identifier2")
var param2 = Type2()

@method(component="componentName", configurations={"genericConfigurationMethodName"},
      method="genericMethodName")
```

Code généré sur iOS (Swift)

```
@method(component="componentName", configurations={"genericConfigurationMethodName"},
      method="genericMethodName")
var component = ClassName()
component1.iOSConfigurationMethodName()
component1.iOSMethodNameArg1(param1, arg2:param2)
```

FIGURE 7.4 – Transofmration d'une annotation ***@method*** vers un code source swift

par exemple le cas entre la définition d'une méthode pour Android et iOS.

— Définir les règles de transformation d'une annotation ***@method*** en code natif par rapport à l'interface complémentaire.

— Définir les règles de compilation contextuelles pour cette nouvelle plate-forme.

Dans cette liste de tâches, nous remarquons que nous ne touchons pas à notre langage commun et l'interface publique des composants. Les développeurs utilisant déjà notre solution n'auront donc pas de temps d'adaptation à notre solution sur cette nouvelle plate-forme. De plus, nous ne sommes pas obligés de ré-implémenter tous les composants existant le plus rapidement possible. En effet, dans notre architecture, nous autorisons les utilisateurs à im-

plémenter l'ergonomie de leurs applications et tout ce qui n'existe pas dans nos composants de façon native. Ainsi, si nous n'avons pas encore ré-implémenter tous les composants existants au moment de la sortie de cette nouvelle plate-forme, les développeurs d'applications mobiles pourront quand même utiliser notre solution avec une liste de composants pouvant être réduite. Ils ne seront pas bloqués pour autant.

Nous pouvons considérer que notre solution est suffisamment flexible et indépendante de toutes les plates-formes mobiles pour prendre en compte les évolutions futures du mobile. En effet, nous pouvons prendre en compte un nouveau langage de programmation mobile dans notre compilateur très rapidement sans pour autant modifier l'architecture de notre solution. C'est un point fondamental pour nous. Nous ne voulons pas reproduire le cas de certaines solutions existantes qui ont mis plusieurs années avant de prendre en compte certaines plates-formes. C'est, par exemple, le cas de Titanium mobile qui a réussi uniquement aux bout de deux ans à mettre en place BlackBerry dans les plates-formes cibles de leur outil.

7.5 Conclusion

Dans ce chapitre, nous avons présenté la base du langage universel de notre solution et son compilateur associé. Notre langage universel est basé sur les annotations et a la particularité de s'intégrer dans n'importe quel langage de programmation. Nous pouvons l'intégrer dans le Java pour Android, l'Objective-C ou le Swift pour iOS, le C# ou .Net pour Windows Phone 8 .... D'ailleurs les annotations sont déjà présentes pour le développement Android, iOS et windows phone 8. En choisissant de se baser sur un langage de programmation existant, le temps d'apprentissage de notre langage sera moindre par rapport à l'apprentissage d'un nouveau langage avec sa propre syntaxe et sémantique. Pour répondre à nos besoins, nous avons détourné l'utilisation habituelle de ce langage. Classiquement, les annotations servent à ajouter des métadonnées à l'objet annoté. Ici, nous l'utilisons comme un langage de programmation classique. L'utilisateur peut avec nos annotations lancer des instructions. Pour permettre l'intégration de composants multiplateformes, nous avons créé trois nouvelles annotations qui sont communes à n'importe quelle plate-forme cible. Elles auront la même syntaxe mais aussi la même sémantique. En effet, la même instruction sur différentes plates-formes aura le même effet sur chacune d'elle : invoquer une fonctionnalité d'un composant. Ainsi, pour l'intégration d'un composant multiplateforme sur plusieurs systèmes d'exploitation cibles, le code à écrire avec nos annotations sera le même sur toutes ces plates-formes. Nous cachons alors toutes les spécificités des différentes plates-formes mobiles. Pour arriver à ce résultat, les instructions écrites avec nos annotations ne contiennent que des références aux interfaces publiques des composants multiplateformes à intégrer. Pour rappel, ces interfaces sont indépendantes de toutes les plates-formes mobiles.

Notre objectif final est de fournir des applications 100% natives. C'est pour cela que nous fournissons en plus de notre langage commun le compilateur associé à ce langage. Ce n'est pas un simple compilateur, il doit, en effet, gérer plusieurs langages de programmation en sortie tel que le Java pour Android ou l'Objective-C pour iOS. La transformation d'une annotation en code natif est basée sur l'interface complémentaire des composants à intégrer. Le compilateur transforme donc uniquement la représentation XML d'une méthode en code natif à partir des éléments fournis par l'intégrateur. Dans l'interface complémentaire des composants, il se peut qu'il y ait des spécificités (plusieurs méthodes natives, paramètres non fonctionnels). Pour gérer ces spécificités, nous avons intégré dans notre compilateur la notion de contexte de compilation. Ce contexte permet au compilateur de déduire tous les paramètres spécifiques à une plate-forme. Ainsi, avec ces déductions couplées à notre langage, nous cachons toutes les spécificités de toutes les plates-formes cibles.

Dans la prochaine partie de cette thèse, nous présentons l'implémentation de notre framework de développement multiplateforme pour mobile avec tous les outils qui simplifient le développement et l'intégration de composants (générateurs d'interfaces XML etc.). Nous avons ensuite implémenté une application entière à partir de composants. À partir de cette application, nous avons effectué une étude sur les gains apportés par notre solution en matière de lignes de code source gagnées, de performances . . .ainsi que sur les limites de notre solution. Enfin, nous avons comparé ces résultats à trois technologies existantes : PhoneGap, Titanium mobile et Xamarin.

Troisième partie

Mise en oeuvre et expérimentations

Chapitre 8

Mise en oeuvre : COMMON

Ce chapitre est consacré à la réalisation des logiciels permettant la mise en oeuvre de l'architecture de notre solution : COMMON. Nous présentons notamment trois logiciels : un générateur d'interfaces publiques et complémentaires, un interpréteur d'interfaces et le compilateur permettant la transformation du code source écrit avec notre langage en code source natif. Ces logiciels seront utilisés pour le développement d'une application présentée dans le prochain chapitre.

Sommaire

8.1 Introduction 115
8.2 Une base commune 116
8.3 Quelques outils pour faciliter l'utilisation de COMMON 118
8.3.1 Faciliter le développement des composants multiplateformes 118
8.3.2 Faciliter l'intégration des composants multiplateformes 123
8.4 Réalisation du compilateur source à source 128
8.5 Charge de travail 131
8.6 Conclusion 132

8.1 Introduction

Dans les chapitres précédents, nous avons détaillé l'architecture de notre solution. Dans cette nouvelle partie, nous la mettons en oeuvre à travers un framework de développement nommé COMMON pour "Component Oriented programming for Mobile Multi Os iNtegration". Actuellement, notre solution a été réalisée pour Android et iOS. Nous nous sommes limités à ces deux plates-formes car ce sont les plus populaires sur le marché. Elles représentent à elles deux 90% du marché. De plus, comme nous l'avons vu dans le chapitre 3, section 3.2, les architectures d'Android et iOS sont très éloignées. Ainsi, si nous arrivons à proposer notre solution pour ces deux plates-formes, nous pensons qu'il sera aussi possible de le faire sur les autres plates-formes mobiles actuelles ou futures. En effet, notre solution est extensible à tout moment.

Dans ce chapitre, section 8.2, nous nous focalisons sur la réalisation de trois logiciels de notre solution. Les deux premiers facilitent le développement et l'intégration de composants

multiplateformes. Le troisième est le compilateur source à source qui s'occupe de transformer les annotations présentes dans un code source natif Android ou iOS en langage natif. Ensuite, section 8.3, nous détaillons les logiciels permettant la génération et l'interprétation des interfaces publiques et complémentaires d'un composant. Nous fournissons un exemple concret d'utilisation avec le composant "HttpRequestManager". Puis, section 8.4, nous présentons la réalisation de notre cross-compilateur et plus particulièrement les interactions entre le compilateur et les utilisateurs. Enfin, section 8.5, nous faisons un bilan sur la réalisation de notre framework de développement.

8.2 Une base commune

Avant de détailler les solutions implémentées, il faut savoir que COMMON et tous les logiciels qui gravitent autour sont multiplateformes. Nous considérons que les développeurs de composants peuvent effectuer l'implémentation de leurs composants autant sur Windows 8 que sur Mac OS X. En effet, en fonction des plates-formes cibles du composant qu'ils veulent réaliser, ils se retrouvent à développer sur l'un ou l'autre des systèmes d'exploitation. De la même façon, les utilisateurs de notre solution pourront intégrer nos composants dans un code source iOS sur Mac OS X ou Windows Phone sur Windows 8. Les logiciels d'aide à l'intégration et notre cross-compilateur doivent donc être aussi capables de fonctionner sur les deux plates-formes. Ici, nous ne nous focalisons pas sur Android car les composants Android ou les applications Android peuvent être implémentés sur Windows 8 et sur Mac OS X. Pour rendre multiplateforme nos logiciels, nous avons choisi de les développer en Java. Il est alors possible de les exécuter sur n'importe quel système d'exploitation de bureau.

Pour le moment, tous les logiciels ont été implémentés pour Android et iOS. Pour chacune de ces deux plates-formes, il nous fallait un parser implémenté en JAVA capable de parser un code source écrit avec le langage de programmation cible (Java pour Android, Objective-C pour iOS). Pour parser le JAVA, nous utilisons : JavaParser [1]. Ce parser s'utilise avec le design pattern Visitor [Mar03]. Ainsi, lorsqu'il parse un fichier Java, pour chaque classe, méthode, déclaration de variable etc. qu'il visite, nous recevons un évènement. Dans notre cas, nous récupérons uniquement les évènements qui sont intéressants pour nous sans nous soucier de tout l'arbre syntaxique trouvé. De plus, ce logiciel n'est pas un simple parser. Il nous permet aussi de modifier une partie de l'arbre syntaxique et ensuite de l'écrire dans un fichier. Cette fonctionnalité est très utile pour la génération de code source. Enfin, il prend déjà en compte le parsing des annotations. Nous n'avons donc pas besoin de modifier le parser pour prendre en compte nos annotations. Quant au parsing de code source Objective-C, nous utilisons : ANTLR [Par13]. Cet outil nous permet de générer un parser en fonction de la grammaire d'un langage de programmation. Une grammaire simplifiée de l'Objective-C est d'ailleurs déjà présente dans les exemples fournis avec l'outil [2]. Cependant, contrairement à JavaParser, l'outil nous retourne directement l'arbre syntaxique lu. C'est ensuite à nous de le parcourir pour récupérer les informations qui nous concernent. Les annotations, comme nous les avons décrites dans le chapitre 7, n'existent pas en Objective-C. Nous avons donc dû modifier la grammaire de l'Objective-C pour y intégrer notre langage. ANTLR est donc très flexible pour la gestion de nouveaux langages. Il suffit d'ajouter nos instructions dans la grammaire cible [BP08]. Dans la suite de nos travaux, nous préconisons son utilisation pour parser du code source Windows Phone 8. Les différences entre les deux parsers sont résumées dans le tableau 8.1.

1. JavaParser : https ://code.google.com/p/javaparser/
2. Grammaire Objective-C pour ANTLR : https ://github.com/antlr/grammars-v4/blob/master/objc/ObjC.g4

TABLE 8.1 – Différences entre les parsers JavaParser et un parser généré avec ANTLR

Parsers	Récupération d'informations dans l'arbre syntaxique	Traitement des annotations	Génération de code	Évolutions possibles de notre langage
JavaParser	à base d'évènements pendant le parsing	déjà intégré	Modification de l'arbre syntaxique puis écriture dans un fichier	Prise en compte uniquement si ces évolutions suivent les spécifications JAVA
Objective-C Parser généré avec ANTLR	Manuellement à la fin du parsing, il faut parcourir l'arbre syntaxique et récupérer les informations voulues	Modification de la grammaire Objective-C	Non supporté	Possibilité d'intégrer des évolutions en modifiant la grammaire du langage à parser

En plus des librairies de parsing, tous les logiciels sont développés à partir de plusieurs librairies communes JAVA que nous avons implémentées. Une de ces librairies contient toute la description informatique des méthodes d'entrée, de configuration et de sortie qui se trouvent dans l'interface publique et l'interface complémentaire d'un composant. Une autre permet de lire les interfaces pour les transformer en modèle informatique. Nos logiciels vont se baser sur ce modèle, soit pour générer ces interfaces, soit pour les interpréter. Parmi nos librairies, il y en a une qui s'occupe exclusivement d'explorer un projet Android ou Objetivce-C. Elle permet de récupérer, entre autres, la liste des classes d'un projet et la structure qu'elles ont entre elles. Ces librairies communes nous permettent de maintenir et de faire évoluer plus facilement nos logiciels. En effet, dès lors que nous ferons une modification sur une de ces librairies, ces modifications seront mises à jour dans tous nos logiciels. Il nous fallait absolument suivre une architecture modulaire pour la prise en compte rapide des évolutions du domaine mobile.

Maintenant que la base de nos programmes est présentée, dans la suite nous allons présenter en détail chacun de nos logiciels.

8.3 Quelques outils pour faciliter l'utilisation de COMMON

La base de notre solution est la description des composants à travers leurs interfaces publiques et complémentaires. Cependant, la génération et l'interprétation de ces interfaces restent laborieuses pour une personne, même confirmée. Par conséquent, nous avons réalisé deux logiciels. Le premier permet aux développeurs de composants de générer automatiquement les interfaces publiques et complémentaires de leurs composants. Le second lui, interprète pour les intégrateurs les interfaces des composants qu'ils veulent insérer dans leurs projets Android ou iOS.

8.3.1 Faciliter le développement des composants multiplateformes

Dans le chapitre 6, nous avons décrit toutes les étapes à suivre pour développer un composant multiplateforme. Pour illustrer nos propos, nous avons fourni un exemple de composant : le "HttpRequestManager". Ce composant est plutôt léger. Il ne contient que trois méthodes d'entrée et aucune méthode de configuration. Cependant, son implémentation est relativement lourde et plus particulièrement la génération des interfaces complémentaires et publiques. Pour rappel, pour développer ce composant, nous avons suivi plusieurs étapes :

i) **Concevoir le composant indépendamment d'une plate-forme cible.** Pour cela le développeur de composant doit lister les méthodes de configuration, d'entrée et de sortie du composant.

ii) **Re-transcrire toutes les méthodes du composant de façon native** à travers des interfaces pour chacune des plates-formes cibles du composant.

iii) **Implémenter le composant sur chacune des plates-formes cibles de façon native.**

iv) **Retranscrire les interfaces natives de chaque plate-forme dans l'interface complémentaire du composant.**

v) **Retranscrire indépendamment d'une plate-forme cible l'interface complémentaire dans l'interface publique du composant.**

Les deux dernières étapes sont primordiales pour l'utilisation des composants dans COMMON. Si ce n'est pas fait ou si l'une des interfaces est malformée alors le composant sera inutilisable dans notre solution. Dans le cas du "HttpRequestManager", la retranscription était plutôt simple même si l'interface complémentaire contient finalement 188 lignes de code XML. Imaginons maintenant qu'un composant contienne plusieurs dizaines de méthodes de configuration, plusieurs dizaines de méthodes d'entrée et autant de méthodes de sortie. L'interface complémentaire contiendra alors des centaines de lignes voire quelques milliers. Dans ce cas, il n'est plus possible de réaliser cette interface manuellement. Pour faciliter cette partie du développement de composants multiplateformes, nous avons automatisé la génération de ces interfaces grâce à un logiciel. Ainsi, le développeur de composants pourra uniquement se focaliser sur l'implémentation de son composant pour plusieurs plates-formes mobiles et non sur cette génération complexe à effectuer.

L'objectif du logiciel est de générer automatiquement les interfaces complémentaires et publiques des composants. Il est donc exclusivement destiné aux développeurs de composants. Ce processus se fait en trois étapes :

i) Renseigner les informations nécessaires à la création des interfaces du composant :
   — La description du composant.
   — La localisation des implémentations natives du composant.

ii) Sélectionner les interfaces natives qui seront exposées à travers l'interface du composant.

iii) Lier les interfaces et les méthodes natives de chaque plate-forme avec celles des autres plates-formes.

Ce sont en fait les étapes que nous avons dû suivre lors de la création des interfaces du composant pour le "HttpRequestManager", chapitre 6, sections 6.2.3, 6.4.3 et 6.5.2.

Renseigner les informations nécessaires à la création des interfaces du composant

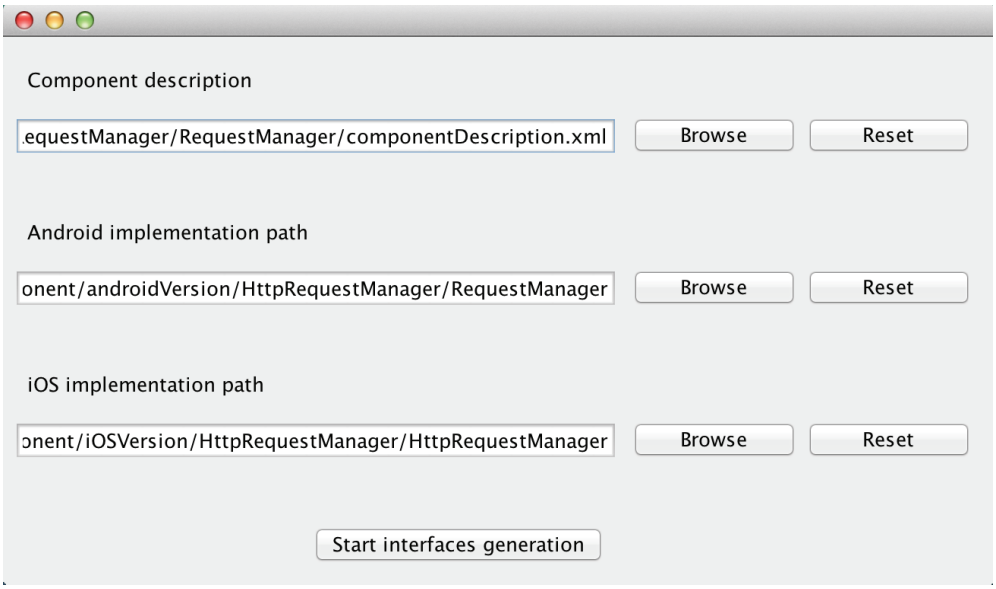


FIGURE 8.1 – Choix des implémentations du composant dont les interfaces doivent être générées

Pour générer les interfaces XML d'un composant, le logiciel se base sur un fichier de description ainsi que sur les implémentations natives du composant (projets android, iOS), figure 8.1. Le fichier de description contient les informations du composant (nom du composant,

version, description ...) sous format XML, voir code 8.1. Ce sont les informations qui seront inscrites dans l'interface publique et complémentaire du composant dans la balise *description*. Quant à elles, les implémentations natives contiennent toutes les méthodes de configuration et d'entrée des composants sur chacune des plates-formes cibles.

Code 8.1 – Description du composant HttpRequestManager

```xml
<?xml version="1.0" encoding="UTF-8"?>
<componentDescription>
    <componentName>HttpRequestManager</componentName>
    <version>1.0</version>
    <date>10/05/2013</date>
    <description>Send http requests</description>
    <vendor>keyneosoft</vendor>
    <dependancies>
        <dependance componentName="DeviceInfoManager"/>
    </dependancies>
</componentDescription>
```

Sélection des interfaces natives qui seront exposées à travers l'interface du composant

Après la sélection des implémentations du composant, le logiciel liste toutes les interfaces natives trouvées dans les projets Android et iOS, figure 8.2. Les développeurs de composants doivent alors sélectionner les interfaces natives publiques du composant. Ce sont elles qui contiennent les méthodes d'entrée et de configuration du composant. En plus de sélectionner les interfaces natives, ils doivent sélectionner les classes qui implémentent ces interfaces.

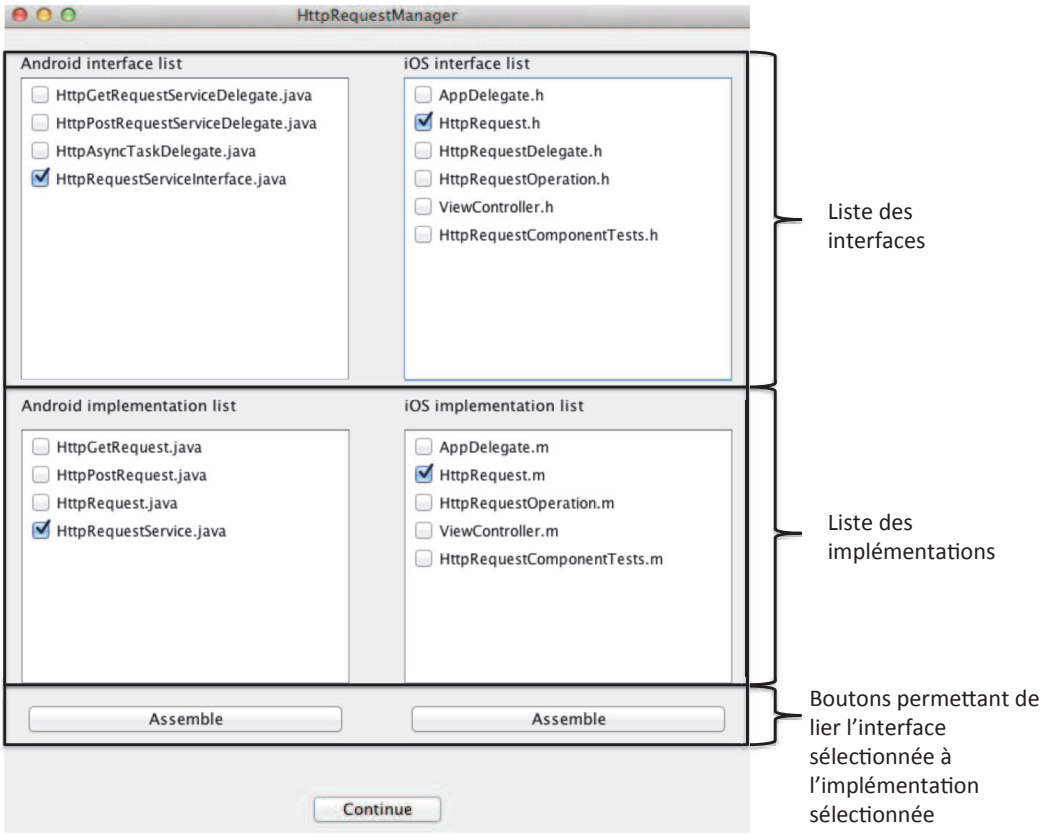


FIGURE 8.2 – Sélection des interfaces natives publiques du composant HttpRequestManager

Liaison des interfaces et des méthodes natives de chaque plate-forme avec celles des autres plates-formes

Ensuite, sur la troisième page du logiciel, figure 8.3a, l'intégrateur doit lier les différentes interfaces natives sélectionnées entre elles. Dans notre exemple, nous lions l'interface android HttpRequestServiceInterface.java et l'interface iOS HttpRequest.h. Toutes les interfaces qui n'auront pas été liées ne seront disponibles que pour la plate-forme en question (Android ou iOS). Après la liaison des interfaces natives, le développeur de composants doit réunir chacune des méthodes de ses interfaces, figure 8.3b. Si une méthode Android est réunie avec une méthode iOS, à la génération des interfaces du composant, elles ne formeront qu'une méthode dans l'interface publique et complémentaire du composant. Si une méthode ne doit pas apparaitre dans l'interface commune elle peut être exclue de la génération. Enfin, toutes les méthodes qui n'auront pas été réunies ou exclues seront considérées comme des méthodes disponibles pour une seule plate-forme. Pour rappel, dans le chapitre 6, section 6.4.3, nous avions effectué cette manipulation manuellement.

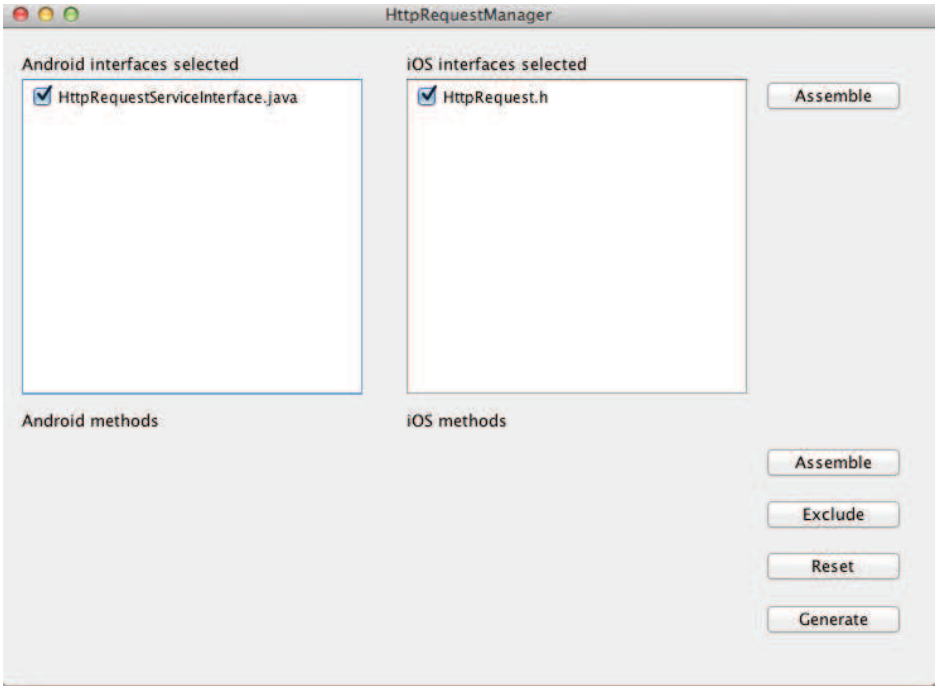
Pour faciliter la génération des interfaces du composant "HttpRequestManager", nous avons modifié les noms des deux méthodes ***sendHttpGetRequest()*** et ***sendHttpPostRequest()*** par ***sendHttpRequest()*** sur Android. Le logiciel les a alors assemblé automatiquement sous une seule méthode Android car elles ont le même nom et le même nombre de paramètres. Après cette étape, les interfaces sont générées.

Pour récapituler, le développeur de composant peut avec ce logiciel sélectionner les interfaces natives qui représentent les méthodes de configuration (doit commencer par *init*) et d'entrée du composant. Ensuite, il permet de réunir ces interfaces ainsi que les méthodes qui les composent. Les méthodes de sortie sont déduites automatiquement par le logiciel.

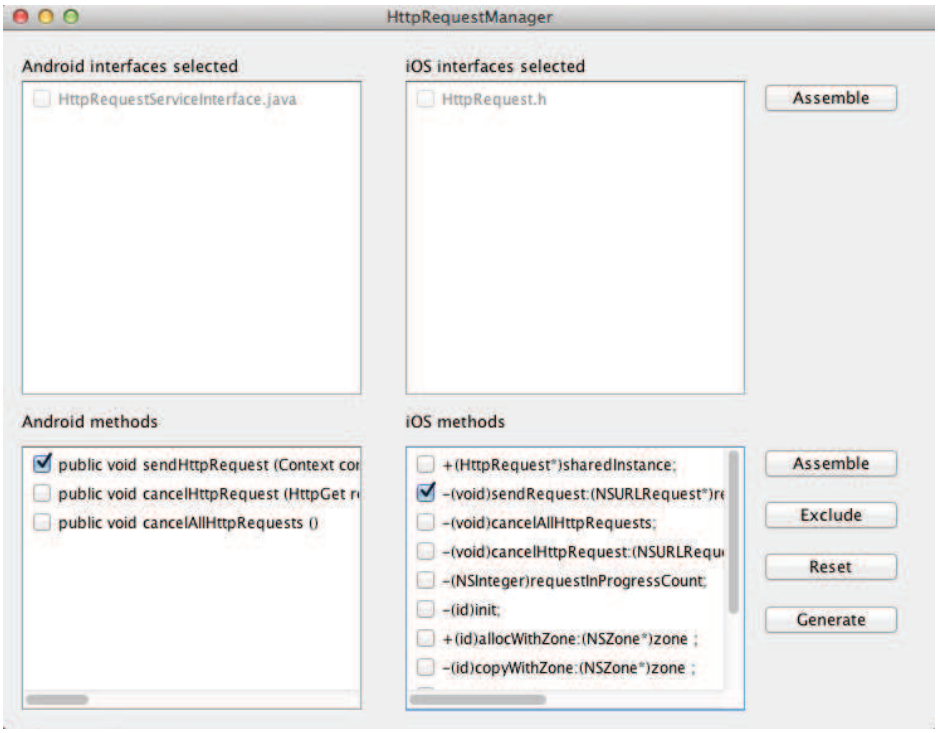
Nous remarquons que certaines informations ne sont pas remplies à travers ce logiciel. C'est le cas de la description des méthodes et de leurs paramètres d'entrée ou de sortie. C'est aussi le cas des types de chaque paramètre : fonctionnel ou non. Ces informations sont remplies par le développeur de composants directement dans son code source à travers ses commentaires. Nous avons choisi ce système car, pour nous, il est trop lourd, lors de l'utilisation de notre logiciel, d'entrer pour chaque méthode sa description et ensuite de réitérer pour chacun des paramètres avec en plus le choix du type de paramètre. Cela devient surtout problématique lorsque le composant a des dizaines de méthodes d'entrée et de configuration. De plus, à chaque modification de son composant et, par conséquent, à chaque fois qu'il génère les interfaces publiques et complémentaires, il devrait refaire la manipulation. En choisissant ce système, il suffit pour les développeurs de composants d'entrer une seule fois ces informations. De plus, il est très simple pour notre logiciel de récupérer ces informations. Nous devons uniquement parser les commentaires liés aux méthodes qui ont été sélectionnées par le développeur de composants.

À travers la figure 8.4, nous montrons un exemple de commentaires valides pour notre logiciel. Ces commentaires, sous format javadoc, sont liés à une des méthodes d'entrée du composant "HttpRequestManager" sous Android. Chaque méthode d'entrée, de configuration et de sortie doit être commentée de cette manière. Dans notre exemple, la description du paramètre *request* est précédée du mot-clé FUNCTIONNAL en rouge. Il sera donc considéré comme de type fonctionnel par notre logiciel. S'il ne l'avait pas été, il aurait été considéré comme un paramètre non fonctionnel. Le texte surligné en jaune correspond à la description de la méthode et à la description du paramètre "request".

Pour récapituler, ce logiciel permet, à travers les informations entrées dynamiquement et les commentaires présents dans les différentes implémentations du composant, de retranscrire



(a) Liaison des interfaces natives provenant de plusieurs plates-formes mobiles différentes



(b) Liaison des méthodes natives provenant de plusieurs plates-formes mobiles différentes

FIGURE 8.3 – Liaisons des différentes implémentations natives d'un composant

```
/**
* Enables to cancel an http get request
* @param request FUNCTIONAL request to cancel
*/
public void cancelHttpRequest(HttpGet request);
```

FIGURE 8.4 – Commentaires de la méthode ***cancelHttpRequest()***

les méthodes d'entrée, de configuration et de sortie du composant en XML. Il va notamment mettre en oeuvre pour le développeur de composants les différentes règles de transformations que nous avons décrites dans le chapitre 6, section 6.5.1. Ainsi, nous facilitons le développement des composants multiplateformes et plus particulièrement le caractère multiplateforme de chaque composant. À la fin du processus, un composant est distribué dans un dossier ayant comme titre le nom du composant. Ce dossier contient lui-même trois dossiers :

— Interfaces : contient l'interface publique et l'interface complémentaire du composant.
— Android : contient l'implémentation du composant sous format jar.
— iOS : contient l'implémentation du composant sous format framework.

Ici, nous ne nous focalisons pas sur le processus de transformation du code source d'un composant Android sous format jar ou d'un composant iOS sous format framework. Ce sont des processus classiques dans la programmation mobile. Pour finir, sur le poste de travail des intégrateurs de composants, nous avons un dossier contenant les composants multiplateformes. C'est à partir de ce dossier que notre compilateur peut sélectionner les composants à intégrer dans le code source natif des intégrateurs.

(a) Contenu du dossier de distribution

(b) Contenu du dossier Android

(c) Contenu du dossier interfaces

(d) Contenu du dossier iOS

FIGURE 8.5 – Dossier de distribution du composant "HttpRequestManager"

8.3.2 Faciliter l'intégration des composants multiplateformes

L'intégration de composants multiplateformes passe par la compréhension des interfaces publiques des composants et par l'apprentissage de notre langage. Pour faciliter ces deux phases, nous avons implémenté une application décrivant ces deux aspects de l'utilisation de notre framework de développement COMMON.

Compréhension des interfaces publiques des composants

La lecture et la compréhension des interfaces XML et plus particulièrement de l'interface publique des composants multiplateformes peut paraître peu abordable pour les intégrateurs de composants. C'est d'ailleurs le cas. L'interface peut, en effet, contenir des centaines de lignes en fonction de la complexité du composant. Par conséquent, il était indispensable pour nous d'implémenter un logiciel permettant l'interprétation de ce type d'interfaces. Le logiciel que nous présentons dans cette section, liste pour l'intégrateur de composants, les méthodes de configuration, d'entrée et de sortie du composant d'une façon beaucoup plus abordable. En plus de cette fonctionnalité, le logiciel guide l'utilisateur pendant l'intégration des composants. Il génère pour lui les annotations qu'il doit intégrer dans son code source Android et iOS dans le but d'utiliser les fonctionnalités des composants.

Ce genre de logiciel existe déjà pour la lecture des fichiers WSDL qui représentent les web services d'un serveur. Les fichiers WSDL sont effectivement très difficiles à lire et à interpréter. Il fallait donc un moyen de les interpréter facilement. SoapUI [3] fait, par exemple, partie de ces logiciels. Il permet à partir d'un WSDL de présenter à un utilisateur tous les web services présents d'un serveur. En plus de cette fonctionnalité, il est possible, à partir de ce logiciel, d'appeler directement les web services en question.

À partir du logiciel, l'intégrateur peut sélectionner le dossier de distribution d'un composant, figure 8.7. Le logiciel interprète alors le contenu de l'interface publique et l'affiche, figure 8.6. Le logiciel affiche la description du composant (cadre bleu). Ces informations se trouvent dans la balise *description* de l'interface publique. Ensuite, il affiche les dépendances, encadrée en jaune, et les plates-formes cibles du composant, encadrée en vert. Ces informations se trouvent respectivement dans la balise *dependancies* et *targetPlatforms* de l'interface publique. Dans le cadre rouge, il affiche le nombre de méthodes de configuration et d'entrée. Enfin, dans le cadre bleu, il liste toutes les méthodes d'entrée et de configuration. Lorsque l'utilisateur clique sur une des méthodes, il obtient sa description :

— Le nom de la méthode.
— La description de la méthode.
— La liste des paramètres d'entrée et de sortie du composant.

Dans cette version du logiciel, il manque les informations liées aux délégates et leurs méthodes de sortie. Elles seront ajoutées dans une prochaine version. Toutes les informations affichées sur cette page sont indépendantes d'une plates-forme cible. Cela s'explique par le fait que nous nous basons uniquement sur l'interface publique. Nous facilitons donc la compréhension du composant. Cependant, l'intégrateur est toujours incapable d'intégrer le composant.

Pour ce faire, à partir du logiciel (bouton "More information about the component"), il est possible de récupérer les annotations nécessaires à l'intégration d'une méthode particulière pour Android, figures 8.7a et 8.7b, ou encore pour iOS, figure 8.8.

Pour rappel, sur Android, la méthode ***sendHttpRequest()*** est représentée de deux façons ***sendHttpRequest(Context, HttpGet, HttpGetRequestServiceDelegate*** ou ***sendHttpRequest(Context, HttpPost, HttpPostRequestServiceDelegate***. Le paramètre Context n'est pas un paramètre fonctionnel. Il sera donc déduit automatiquement par le compilateur. Aucune annotation n'est donc nécessaire pour le renseigner. Le deuxième paramètre, lui, est un paramètre fonctionnel. Il faut donc le spécifier avec l'annotation ***@var***, cadres rouges, figure 8.7a.

En plus de fournir l'annotation nécessaire, il donne aussi une indication sur le placement de l'instruction. Elle doit être placée devant un objet de type **HttpGet** ou **HttpPost**. De

3. Site web officiel de SoapUI : http ://www.soapui.org/About-SoapUI/what-is-soapui.html

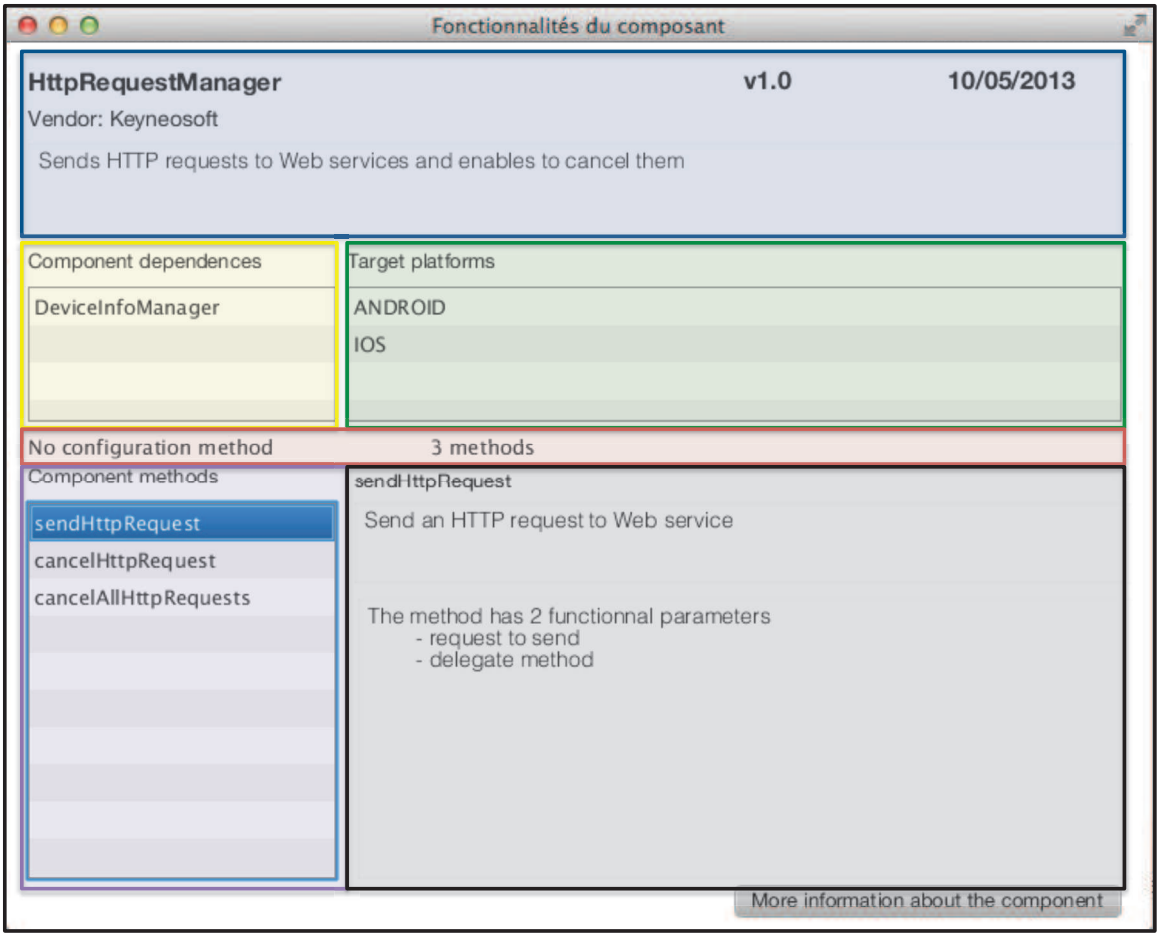


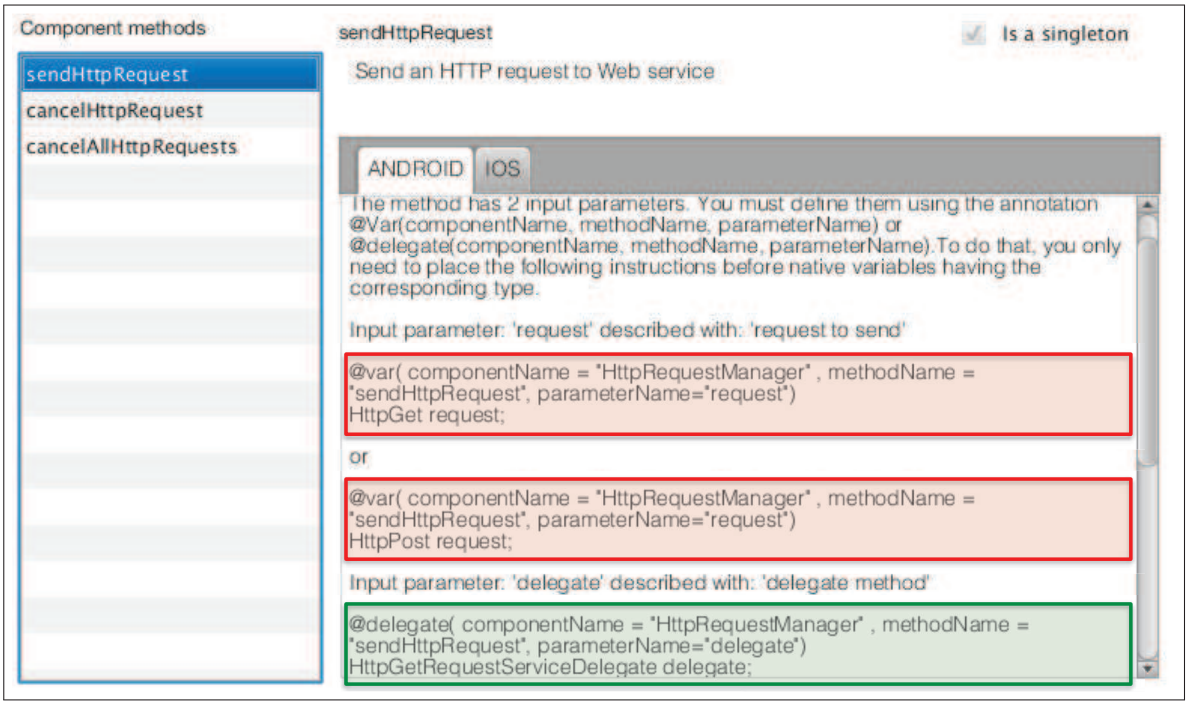
FIGURE 8.6 – Résumé des fonctionnalités du composant "HttpRequestManager"

la même façon, le troisième paramètre est fonctionnel. Il doit donc être spécifié avec une annotation. Vu que c'est un paramètre de type delegate, il doit être décrit avec l'annotation ***@delegate***, encadrée en vert. Bien sûr, comme pour le premier paramètre, le logiciel laisse le choix entre placer l'annotation ***@delegate*** devant un delegate de type HttpGetRequestServiceDelegate ou HttpPostRequestServiceDelegate. La suite du texte, dans l'onglet Android, est présenté dans la figure 8.7b. Il décrit l'annotation ***@method*** nécessaire à l'appel de la méthode ***sendHttpRequest()***.

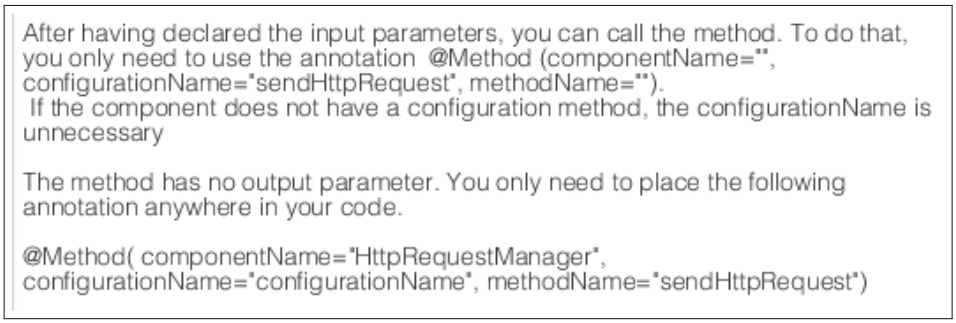
Contrairement à l'intégration Android, sur iOS, figure 8.8, il n'y a pas de dédoublement de la méthode. Ainsi, l'intégrateur doit placer ses annotations devant un seul type d'objet possible. Les déclarations des deux paramètres fonctionnels ont été encadrées en rouge et vert. Nous distinguons aussi le début de la description de l'annotation ***@method*** nécessaire à l'appel de la méthode ***sendHttpRequest***.

Nous sommes obligés de diviser cette partie en fonction des plates-formes cibles. Cela s'explique par le fait qu'en fonction de la plate-forme cible, les annotations ***@var*** et ***@delegate*** ne se placent pas aux mêmes endroits, ou plutôt devant des objets de mêmes types. Nous serons toujours dans ce cas car les langages de programmation cibles sont trop différents. Cependant, nous remarquons toujours que les annotations sont les mêmes, que l'on soit sur Android ou sur iOS. Pour faire cette distinction, le logiciel utilise l'interface complémentaire du composant.

Dans les textes décrivant les annotations à intégrer dans un code source, figures 8.7a, 8.8



(a) Début de l'aide à l'intégration du composant "HttpRequestManager" sur Android



(b) Fin de l'aide à l'intégration du composant "HttpRequestManager" sur android

FIGURE 8.7 – Aide à l'intégration du composant "HttpRequestManager" sur Android

et 8.7b, nous avons à chaque fois donné une explication sommaire sur l'annotation à utiliser. Pour plus d'informations, nous avons créé une section dans le logiciel dédié à l'apprentissage de notre langage.

L'apprentissage de notre langage commun

Dans COMMON, nous n'utilisons pas les annotations de façon classique (appel de méthode) et nous avons créé nos propres annotations. Par conséquent, nous avons décrit toutes les instructions de notre langage à travers notre logiciel, figure 8.9. L'intégrateur peut alors visualiser une description de chaque annotation.

Pour récapituler, avec ce logiciel, il est possible pour n'importe quel utilisateur d'intégrer un composant. Le logiciel liste les méthodes d'entrée et de configuration de celui-ci. Ensuite, il génère les annotations nécessaires à l'intégration de chaque méthode d'entrée. De plus, il permet d'avoir une vue d'ensemble sur notre langage commun. Dans le cadre de la thèse et de mon parcours personnel, il était plus facile et rapide de mettre en oeuvre ce logiciel en JAVA. Par la suite, il sera transposé à travers un site web. Il sera alors plus simple pour les

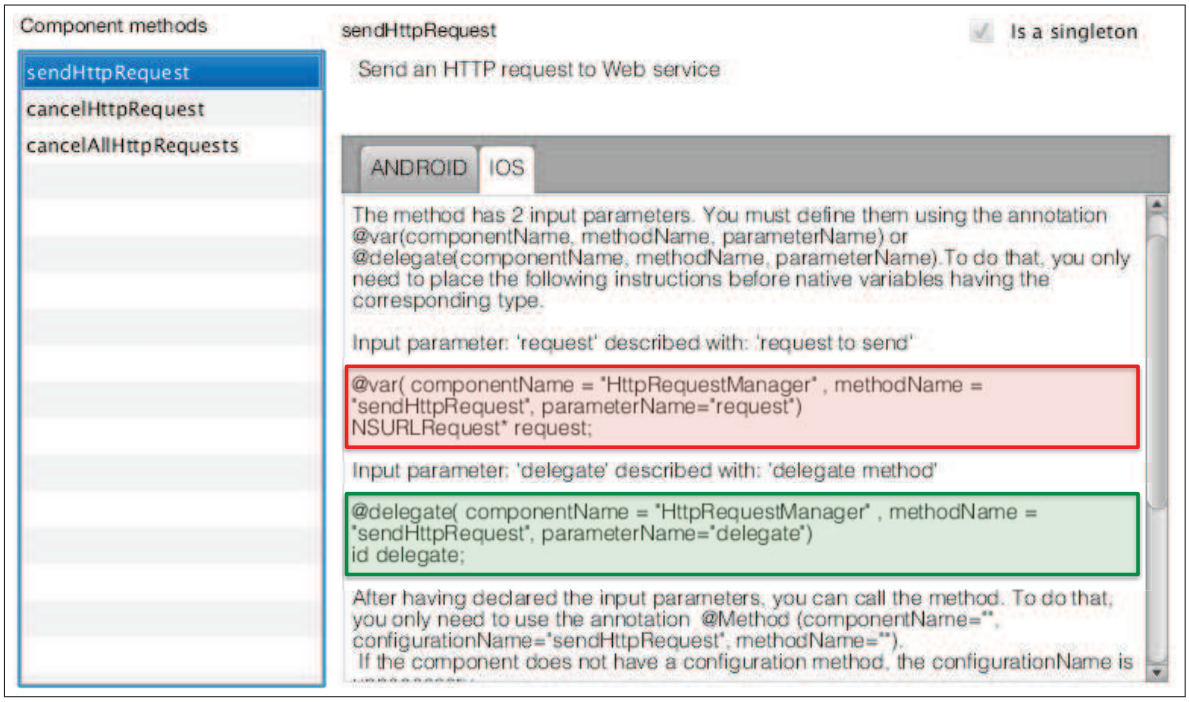


FIGURE 8.8 – Aide à l'intégration du composant "HttpRequestManager" sur iOS

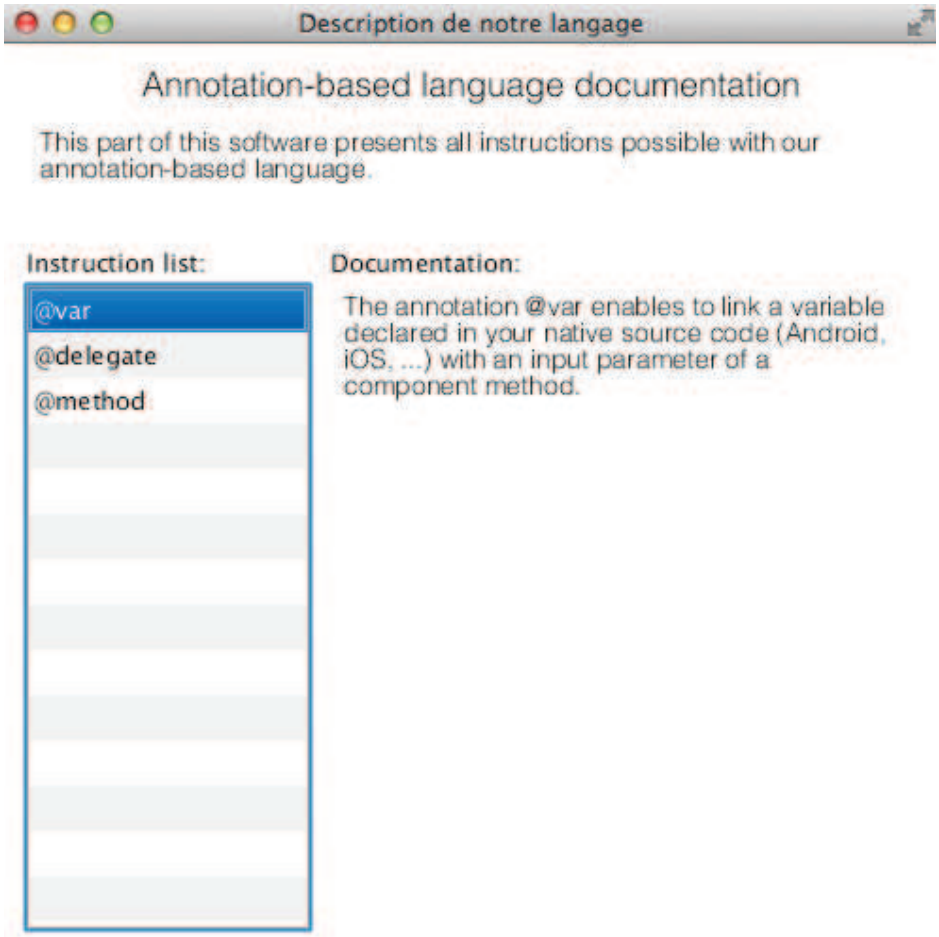


FIGURE 8.9 – Description des annotations de notre langage

utilisateurs d'accéder à la documentation de notre langage et des composants.

8.4 Réalisation du compilateur source à source

Dans cette section, nous n'allons pas reprendre toutes les règles que nous avons déjà décrites dans le chapitre 7, section 7.3.2. Il faut juste savoir qu'elles ont été implémentées dans notre compilateur. Ici, nous allons plutôt discuter de la réalisation et des interactions entre l'intégrateur de composant et le compilateur. Pour rappel, c'est l'intégrateur de composant qui va lancer la compilation de son application.

Tout d'abord, notre compilateur source à source utilise nos librairies communes pour effectuer la transformation des annotations en code source natif. Par exemple, lors de la pré-compilation d'un projet iOS, figure 8.10, le compilateur demande à la librairie "ProjectExplorer" de lister les fichiers de code source présents dans le dossier. Cette librairie est d'ailleurs déjà utilisée dans le logiciel présenté dans la section 8.3.1. Ensuite, pour chacun des fichiers trouvés, le compilateur demande à la librairie Objective-C Parser de décomposer le code source du fichier en arbre syntaxique. À partir de cet arbre syntaxique le compilateur s'occupe de transformer toutes les annotations de type ***@mehtod*** en code source natif. Pour ce faire, il suit l'algorithme présenté à travers le code 8.2. Pour valider qu'une annotation ***@method*** est déclarée correctement, c'est-à-dire que le composant et la méthode appelés existent ainsi que les paramètres d'entrée de la méthode sont spécifiées avec des annotations ***@var*** ou ***@delegate***, nous utilisons encore une librairie commune. Cette librairie fait le lien entre les annotations déclarées et les interfaces publiques et complémentaires du composant. Elle permet aussi de générer le code source associé à une annotation de type ***@method***. Cette librairie est aussi utilisée par notre logiciel présenté dans la section 8.3.2.

Code 8.2 – Algorithme de compilation d'annotations en code source natif

```
1 input variable syntacticTree, filePath
2 FUNCTION compileFile
3
4     var methodAnnotationList = getMethodAnnotationList(syntacticTree);
5     var varAnnotationList = getVarAnnotationList(syntacticTree);
6     var delegateAnnotationList = getDelegateAnnotationList(syntacticTree);
7
8     FOR int i = 0 TO i < methodAnnotationList.length
9         IF isValidMethodAnnotation(methodAnnotationList[i],
              getVarAnnotationList, getDelegateAnnotationList, syntacticTree)
              == true
10            syntacticTree = convertMethodAnnotationToNativeCode(
                  methodAnnotationList[i], syntacticTree);
11        ELSE
12            THROWS error;
13            RETURN false;
14        ENDIF
15    ENDFOR
16
17    write(filePath, syntacticTree);
18
19    RETURN true;
20 ENDFUNCTION
```

Bien sûr, après avoir transformé le code écrit avec nos annotations, le compilateur se charge d'intégrer dans l'application hôte les implantations des composants appelés. Pour ce faire, en fonction du contexte de compilation et plus particulièrement de la plate-forme pour laquelle le compilateur est utilisé, il copie l'exécutable du composant Android ou iOS et le

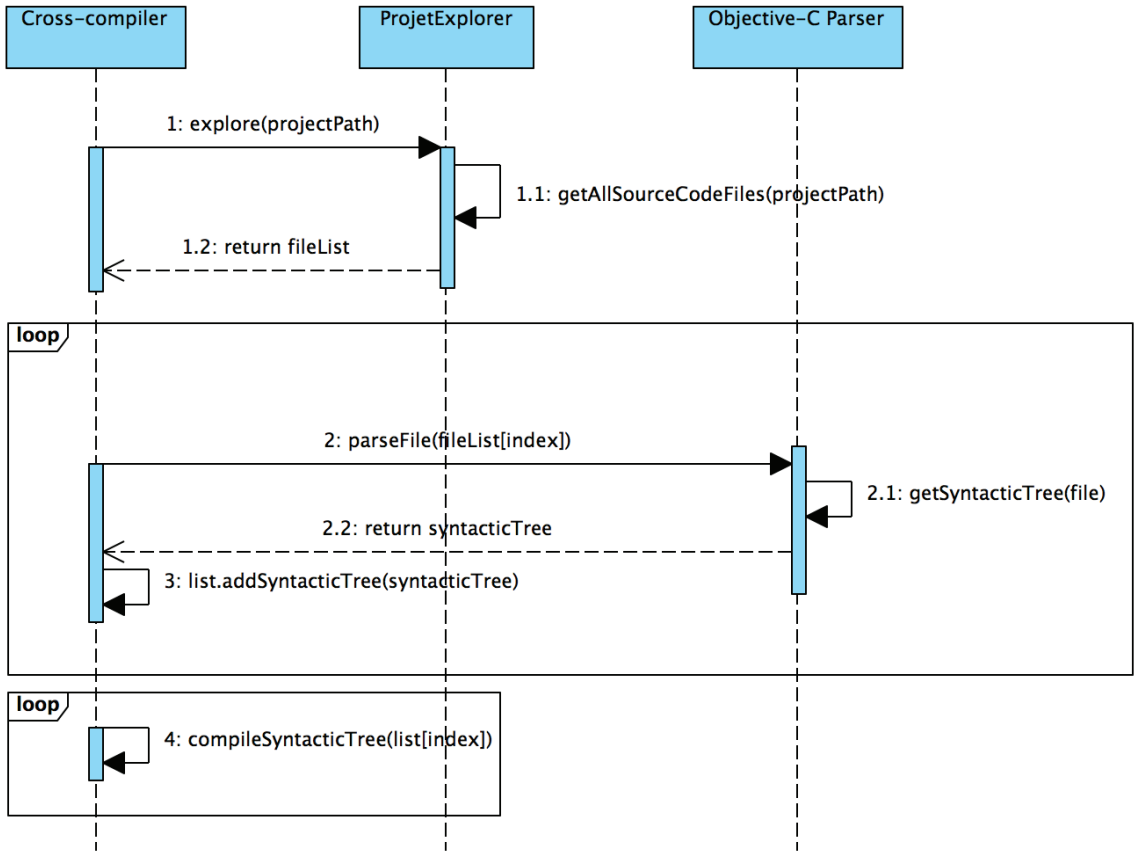


FIGURE 8.10 – Processus de pré-compilation pour un projet iOS

lie au projet cible. De la même façon, il copiera aussi tous les exécutables des dépendances du composant.

Plusieurs types d'erreurs peuvent être détectées avec notre cross-compilateur :

- Les erreurs de déclaration des méthodes liées à l'utilisation de l'annotation ***@method*** :
  - Le composant spécifié n'existe pas.
  - La méthode d'entrée spécifiée n'existe pas.
  - Les méthodes de configuration listées n'existent pas.
  - La variable annotée n'a pas le même type que le paramètre de sortie de la méthode (cas d'une fonction).
- Les erreurs liées aux déclarations des variables avec l'annotation ***@var*** ou ***@delegate*** :
  - Le composant spécifié n'existe pas.
  - La méthode spécifiée n'existe pas
  - Le paramètre d'entrée spécifié n'existe pas pour la méthode souhaitée.
  - La variable annotée n'a pas le même type que le paramètre d'entrée. souhaité
- L'impossibilité de déduire les paramètres non fonctionnels à partir du contexte de compilation de l'application.
- L'impossibilité de déduire la méthode à appeler.

Pour que le compilateur soit adopté par les développeurs, il doit s'intégrer parfaitement dans les environnements natifs de compilations. Ainsi les erreurs doivent s'afficher comme des erreurs détectées nativement par le compilateur officiel. Actuellement ce n'est pas le cas. Notre compilateur doit être lancé par l'intégrateur de composants en ligne de commande, voir code 8.3. La commande prend en entrée le chemin du projet à pré-compiler et le dossier

contenant les composants sous format distribution. Après la pré-compilation, c'est encore à l'intégrateur de lancer la compilation du projet avec le compilateur officiel de la plate-forme ciblée. Ce procédé pose un problème. Il oblige l'intégrateur à interagir avec deux outils différents : le terminal et l'environnement de développement cible (eclipse, XCode ...). Il faudrait donc intégrer le lancement du compilateur dans l'environnement de développement concerné (ou inversement).

Code 8.3 – Commande pour lancer le cross-compilateur

```
1 java -cp CrossPlatformCompiler.jar com.keyneosoft.crossplatformcompiler.
      services.CrossCompiler "projectPath" "componentPath"
```

Ensuite, dans la version actuelle, les modifications effectuées par notre compilateur sont visibles. En effet, l'intégrateur a accès au code source généré par notre compilateur. Il peut même le modifier. Ce n'est pas un problème en soi. Cependant, pour nous le processus de compilation est difficile à gérer. À chaque utilisation de notre compilateur, il faut savoir quelles annotations ont déjà été compilées, lesquelles doivent être compilées, etc. Pour ne pas avoir ces problèmes, dans l'idéal, à la fin du processus de compilation (pré-compilation suivie de la compilation de l'application), le projet devrait être identique au projet avant la compilation. Pour cela, il faut intégrer un processus de post-compilation qui serait lancé à la fin de la compilation avec le compilateur officiel. Ce nouveau processus supprimerait alors tout le code source généré au moment de la précompilation. Sur la figure 8.11, nous avons représenté le processus idéal. Dans nos futurs travaux et l'industrialisation de nos outils, nous fournirons ce système. Dans le cadre de la thèse, le processus de pré-compilation seul était suffisant.

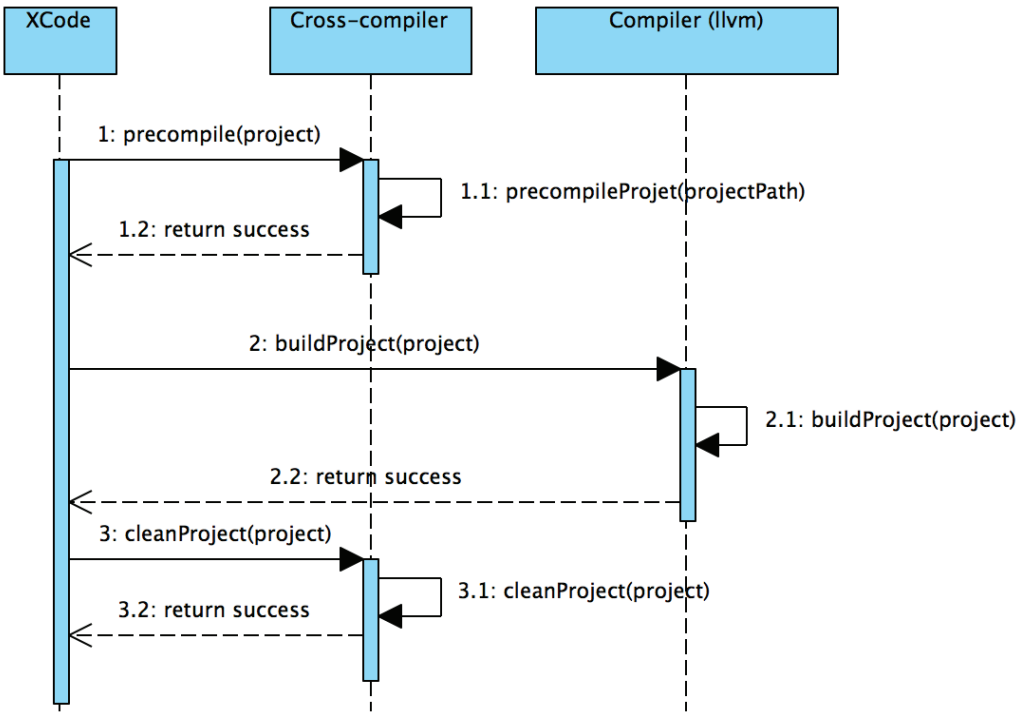


FIGURE 8.11 – Utilisation idéale de notre cross-compilateur

8.5 Charge de travail

Les outils réalisés sont une partie conséquente du travail de thèse. Effectivement, pour arriver à ce résultat, nous avons implémenté plus de 10000 lignes de code réparties en plusieurs librairies communes et trois logiciels, tableau 8.2. Nous avons aussi utilisé des librairies conséquentes pour le projet, ce qui représente plus de 34000 lignes de code, tableau 8.3. Dans le prochain chapitre, nous verrons que pour tester notre solution, nous avons dû encore développer plus de 20000 lignes de code. Cependant, cet effort était pour nous, indispensable. Nous devons réussir à atteindre un niveau d'utilisation très proche de ce que nous voulons réellement. Aujourd'hui, pour terminer les logiciels présentés, dans le but de les utiliser en entreprise, il nous faudrait en terme de lignes de code en écrire au moins le double. Pour autant, l'implémentation actuelle suffit à valider l'approche proposée dans cette thèse. La récupération du nombre de lignes de code a été effectuée avec le plugin eclipse : Metrcis [4].

Table 8.2 – Nombre de lignes de code écrites pour réaliser les logiciels liés à COMMON

Logiciels implémentés	Nombre de lignes de code
Librairies communes	3752
Générateur d'interfaces	2687
Interpréteur de composants	1808
Compilateur	2109
Total	10356

Table 8.3 – Nombre de lignes de code des logiciels utilisés par COMMON

Logiciels utilisés/générés	Nombre de lignes de code
JavaParser	21111
Objective-C Parser généré avec ANTLR	13276
Total	34387

C'est un projet conséquent mais qui n'en est qu'à ses débuts. Nous avons déjà d'ailleurs listé plusieurs évolutions possibles. Pour la perspective de ces évolutions, nous avons développé la base de notre solution à partir de plusieurs librairies communes. Dans un premier temps, nous voulons industrialiser les outils existants et plus particulièrement le compilateur pour Android et iOS. Dans un second temps, nous allons réaliser un site communautaire autour de COMMON. Nous voulons créer un groupe actif de développeurs de composants venant d'entreprises ou non dans le but d'avoir le plus large panel de composants possibles. En proposant un large panel de composants, nous voulons viser l'intégration de nos composants avec notre solution dans les entreprises de développement mobiles. Bien sûr, au fur et à mesure nous intégrerons les évolutions du domaine du développement mobile ainsi que de nouvelles plates-formes.

4. Site web officiel de Metrics : http ://eclipse-metrics.sourceforge.net

8.6 Conclusion

Dans ce chapitre, nous avons présenté la mise en oeuvre de notre solution à travers le framework de développement COMMON pour Android et iOS. Il est composé d'un panel de composants multiplateformes et d'un compilateur permettant de transformer les annotations nécessaires à l'intégration des composants. Pour développer les composants, nous avons réalisé un logiciel qui génère automatiquement les interfaces publiques et complémentaires du composant à partir de ses implémentations natives. De la même façon, nous avons implémenté un logiciel d'aide à l'intégration de composants. Ce logiciel s'occupe pour l'intégrateur d'interpréter les interfaces publiques et complémentaires d'un composant et génère automatiquement les annotations nécessaires aux appels de chacune des méthodes d'entrée.

Pour arriver à ce résultat, nous avons implémenté un total de 10356 lignes de code. Nous estimons que pour avoir une solution qui réponde totalement à nos besoins, il nous faudra au moins le double. Dans le prochain chapitre, nous avons utilisé nos outils en condition réelle pour implémenter et intégrer des composants. Ainsi, nous avons implémenté une application de type utilitaire entièrement avec notre solution. Dans les prochains chapitres, nous comparerons cette application avec la même application implémentée avec d'autres solutions existantes.

Chapitre 9

Évaluation de COMMON

Dans le cadre d'une thèse CIFRE, nous sommes confrontés à des développements réels d'applications. Ainsi, pour évaluer COMMON, nous avons implémenté une application dans des conditions réels de développement qui aurait pu être commercialisée. Pour cela, nous avons fait appel aux graphistes de Keyneosoft. Nous avons implémenté cette application pour Android et iOS de façon native et ensuite avec COMMON. Nous avons finalement comparé les deux versions de l'application sur chacun des systèmes d'exploitation (nombre de lignes de code, performances, consommations de ressources, etc.).

Sommaire

9.1 Introduction 133
9.2 Application d'évaluation 134
9.3 Implémentation de LocaPlace avec COMMON 136
9.4 Évaluations 138
9.4.1 Faisabilité de l'application 139
9.4.2 Diminution de la charge de développement 140
9.4.3 Performances 143
9.4.4 Consommation des ressources 147
9.5 Conclusion 151

9.1 Introduction

Dans le chapitre précédent, nous avons présenté les outils permettant d'utiliser notre framework de développement sur iOS et Android. Dans ce chapitre, nous allons comparer COMMON avec le développement 100% natif (Java pour Android, Objective-C pour iOS). Pour ce faire, nous avons implémenté une application avec notre solution et de façon native pour Android et iOS. La version implémentée avec COMMON intègre cinq composants multiplateformes qui sont tous intégrés avec notre langage commun. Ici, nous ne nous focalisons pas sur les deux implémentations mais sur les résultats obtenus avec notre solution en prenant en compte quatre critères : la faisabilité de l'application, la diminution de la charge de développement, les performances et la consommation de ressources. L'objectif est pour chaque

critère que nous soyons au moins égal au développement natif et pour le critère de diminution de la charge de développement que nous soyons bien inférieur au développement natif. Notre objectif final est de cacher les différences entre plates-formes cibles pour justement diminuer la charge de travail d'une application mobile.

Le chapitre est organisé de la façon suivante : section 9.2, nous présentons l'application utilisée pour notre évaluation. Section 9.3, nous présentons l'implémentation de l'application de test avec COMMON. Pour finir, dans la section 9.4, nous comparons notre solution avec le développement natif.

9.2 Application d'évaluation

Pour évaluer notre solution, il était important pour nous de l'utiliser en condition réelle, c'est-à-dire lors du développement d'une application mobile complète. Pour ce faire, nous avons choisi d'implémenter une application de type utilitaire sur Android et iOS. C'est ce type d'applications qui est habituellement produit par Keyneosoft ainsi que par la plupart des éditeurs d'applications mobiles. Plus particulièrement, nous avons choisi d'implémenter une application permettant la recherche de points d'intérêts autour de soi ou dans une ville particulière. Cette application a été nommée LocaPlace. Pour l'anecdote, au début de la thèse CIFRE j'ai suivi une formation dans le développement iOS qui s'est terminé par l'implémentation d'une application du même type pour un client de Keyneosoft. Sur la figure 9.1, nous montrons les enchainements de vues de cette application à travers un storyboard.

L'application est divisée en deux parties disponibles à partir d'un menu (vue 1) :

— La recherche de points d'intérêts. Dans un premier temps, l'utilisateur entre ses besoins (vue 2). Il sélectionne le type de points d'intérêts qu'il recherche (restaurant, pharmacie, super marché, etc.) et l'endroit où porte sa recherche (autour de soi ou dans une ville particulière). Après ceci, il peut lancer sa recherche, l'application ouvre alors une page contenant une liste de points d'intérêts (nom du lieu, adresse, type du lieu) correspondant à la recherche. Cette liste est affichée de deux façons différentes : sous la forme d'une liste scrollable (vue 3) ou sur une carte (vue 4). À partir de ces deux vues, l'utilisateur peut sélectionner le point d'intérêt souhaité et ainsi avoir plus d'informations sur celui-ci. Dans ce cas, l'application affiche les horaires du lieu, l'adresse du site web, la note que les utilisateurs ont donnée au lieu (vue 5), des photos (vue 6), des commentaires (vue 7). Enfin, l'utilisateur peut visualiser l'itinéraire pour accéder au lieu sélectionné à partir de l'endroit où il se trouve. L'itinéraire est affiché sous deux formats : une liste scrollable (vue 8) ou sur une carte (vue 9).

— La récupération d'informations sur un point d'intérêt particulier à partir d'un QR-Code. À partir de cette partie de l'application, l'utilisateur peut scanner un QR-Code dans lequel sont contenues toutes les informations sur un lieu (vue 10). Le format du texte contenu dans le QR-Code est spécifique à l'application. L'application affiche alors ces informations (nom, adresse ...) sur une nouvelle page (vue 11). L'utilisateur peut ensuite visualiser l'itinéraire pour y accéder à partir de l'endroit où il se trouve (vue 11 et 12).

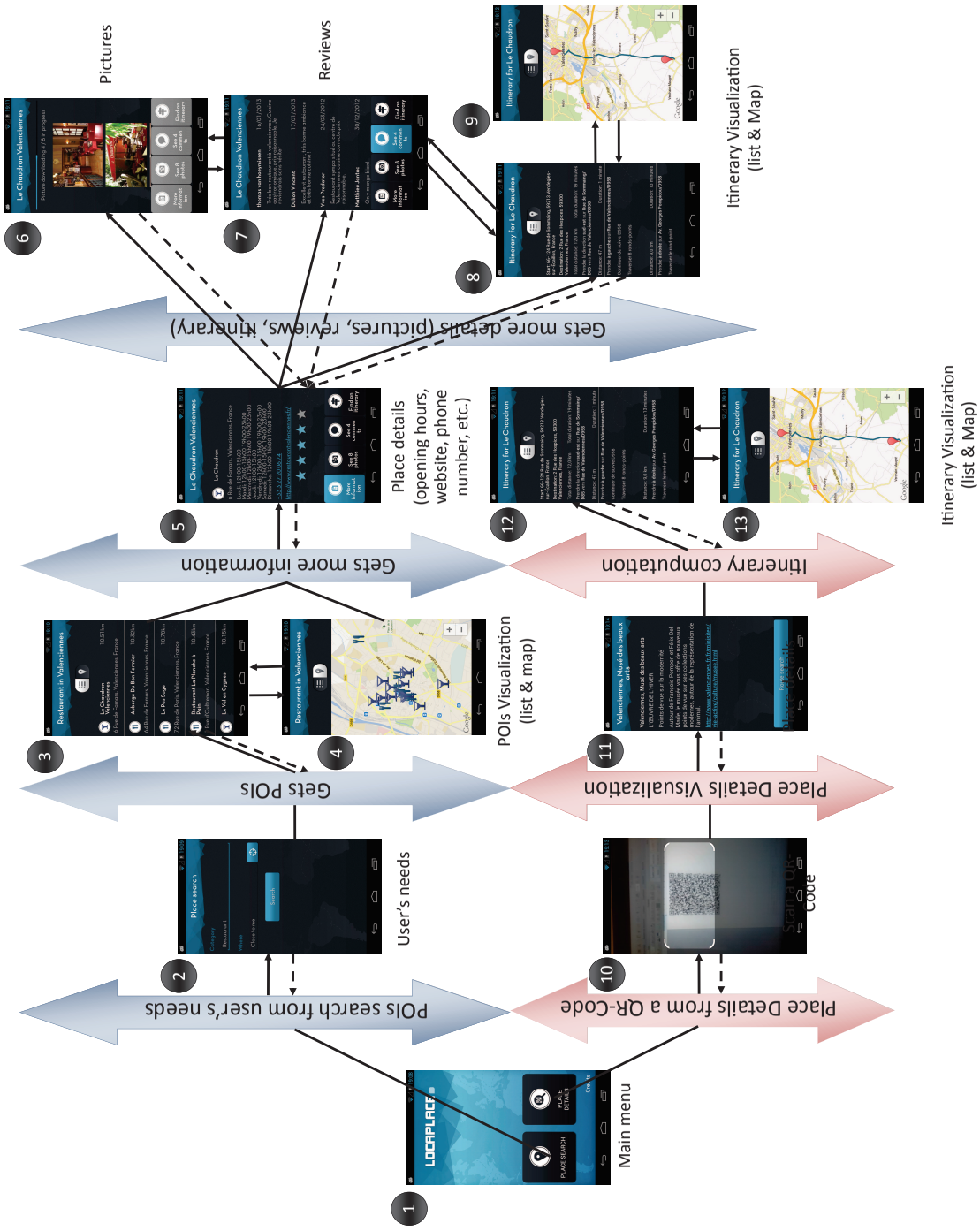


FIGURE 9.1 – Storyboard représentant l'application LocaPlace

Toutes les informations récupérées à travers cette application proviennent de web services fournis par Google. La recherche de lieux passe par l'API Place Search [1]. La recherche des informations supplémentaires sur chaque lieu passe par l'API Place Details [2]. Quant à elles, les photos d'un lieu sont récupérées avec l'API Place Photos [3]. Enfin, la recherche d'itinéraire est effectuée grâce à l'API Google Directions [4].

Le scan de QR-Code passe par une librairie développée par Keyneosoft. Dans la thèse, nous ne la prendront pas en compte pour des raisons de confidentialité. Nous ne récupérerons donc aucun résultat sur cette partie de l'application. Nous ne comptabiliserons pas non plus les lignes de code nécessaires à cette librairie.

Nous avons choisi cette application car elle est représentative des fonctionnalités que l'on pourrait trouver dans n'importe quelle application de type utilitaire :

— Récupération d'informations sur le téléphone (localisation, état des connexions réseau).
— Autocomplétion sur un fichier texte (type de points d'intérêts).
— Autocomplétion sur une base de données (villes).
— Appel à des web services.
— Parsing de réponses JSON (Google Places API).
— Parsing de réponses XML (Google Directions).
— Interaction avec les capteurs du téléphone (caméra).
— Interaction avec des APIs tierces comme Google Map (affichage de lieu et d'itinéraire sur une carte).
— Une interface utilisateur classique (scroll vertical, horizontal, animations lors des changements de page etc.).
— L'application est internationale. Elle fonctionne en français et en anglais.

L'objectif final est, bien-sûr, de montrer que notre solution peut être utilisée pour n'importe quelle application de type utilitaire ayant tous types de fonctionnalités.

Dans le cadre de l'évaluation de COMMON, nous avons implémenté LocaPlace de deux façons. Dans un premier temps, nous avons développé l'application de façon 100% native pour Android et iOS. Nous ne nous focaliserons pas sur cette implémentation. Elle a été effectuée classiquement. Dans un deuxième temps, nous avons implémenté l'application avec COMMON. C'est l'objet de la section suivante.

9.3 Implémentation de LocaPlace avec COMMON

L'implémentation de cette application est divisée en deux parties. Effectivement, notre framework de développement autorise l'implémentation de la structure de l'application de façon native et l'intégration de composants multiplateformes avec notre langage commun. Tout d'abord, nous discutons de la partie correspondant à la structure de l'application et qui a donc été implémentée nativement. Ensuite, nous faisons la liste des composants que nous avons utilisés pour cette application.

1. Site web de l'API Place Search : https ://developers.google.com/places/documentation/search ?hl=FR
2. Site web de l'API Place Details : https ://developers.google.com/places/documentation/details ?hl=FR
3. Site web de l'API Place Photos : https ://developers.google.com/places/documentation/photos ?hl=FR
4. Site web de l'API Google Directions : https ://developers.google.com/maps/documentation/directions/ ?hl=fr

Structure de l'application

Précédemment, nous avons présenté l'application et la liste des fonctionnalités communes à toutes les applications de type utilitaire qu'elle intègre. Elle contient aussi beaucoup de fonctionnalités spécifiques au domaine de l'application. Par exemple, toutes les applications n'affichent pas des itinéraires en bleu. Lors de la présentation de l'architecture de notre solution, chapitre 5, nous avions mis en avant le fait que la structure de l'application ainsi que toutes les spécificités de l'application multiplateforme à réaliser doivent être implémentées en langage natif (Java pour Android, Objective-C pour iOS, etc.). C'est ensuite dans cette structure, autrement dit, ces codes sources natifs, que les intégrateurs pourront intégrer nos composants multiplateformes. Par conséquent, pour la réalisation de l'application LocaPlace, nous avons développé la structure de l'application en langage natif. Ainsi, nous avons dû faire deux projet : un projet Android en Java sous eclipse et un projet iOS en Objective-C sous XCode. C'est dans ces deux projets que nos composants multiplateformes seront intégrés avec notre langage commun basé sur les annotations.

La structure et la partie spécifique de l'application représente :
— L'affichage des données (conception des vues).
— La navigation entre les vues.
— Les animations entre les vues.
— Les interactions entre l'utilisateur et l'application.
— La logique métier : quand est-ce que l'application localise l'utilisateur ? Quand est-ce que l'application récupère les données ? Quand est-ce que les données sont affichées ? C'est en langage natif que nous orchestrons la logique de l'application.
— L'affichage des points d'intérêts sous le format d'une liste ou d'une carte.
— L'affichage d'un itinéraire sur une carte avec un zoom sur la zone de l'itinéraire.
— Le parsing du QR-Code lu.
— ...

Toutes ces fonctionnalités sont donc développées spécifiquement pour cette application en langage natif. Dans la suite, nous listons les composants intégrés.

Les composants intégrés

Dans la section précédente, nous avons listé les fonctionnalités que l'application fournit et qui sont communes à toutes les applications de type utilitaire. Ce sont ces fonctionnalités qui doivent être remplies par nos composants multiplateformes. En effet, elles sont indépendantes d'une application particulière. Elles pourront donc être réutilisées dans d'autres applications.

Nous n'allons pas pour chaque composant donner toutes les étapes de développement (conception, interfaces natives, interface complémentaire, interface publique). Nous l'avons déjà fait pour le composant "HttpRequestManager" dans le chapitre 6, sections 6.2.3, 6.4.3 et 6.5.2. Nous allons plutôt présenter les fonctionnalités de chaque composant à un haut niveau d'abstraction(méthodes de configuration et d'entrée). L'objectif est d'avoir une vue d'ensemble des composants utilisés dans l'application. Pour les implémenter, nous nous sommes basés sur la même méthodologie que pour l'implémentation du composant "HttpRequestManager". Seule la génération des interfaces complémentaires et publiques a été effectuée différemment. En effet, nous avons généré automatiquement les interfaces avec le logiciel présenté dans le chapitre 8, section 8.3.1.

L'application intègre 5 composants :
— **"AutoCompletion"** : ce composant permet d'autocompléter un mot à partir d'un tableau, d'un fichier texte, d'un fichier CSV ou d'une base de données. Le choix de la

source qui servira à l'autocomplétion se fait avec des méthodes de configuration. En méthode d'entrée, le composant permet de rechercher tous les mots qui commencent par un mot ou de rechercher tous les mots qui contiennent un mot particulier. Le composant retourne une liste de mots pour chacune des méthodes d'entrée et fonctionne en mode synchrone. Il ne contient pas de delegate.

— **"CityAutoCompletion"** : ce composant est une spécialisation du composant "AutoCompletion". En effet, ici, l'autocomplétion sur un mot se fait uniquement à partir d'une base de données. De plus, le composant ne retourne pas une simple liste de mots mais des objets représentant une ville (nom de la ville et code postal).

— **"DeviceInfoManager"** : ce composant permet de récupérer des informations sur le téléphone telles que la localisation, l'état du réseau, les capteurs disponibles (NFC, bluetooth ...), etc.

— **"HttpRequestManager"** : ce composant permet l'envoi de requête Http à des web services. Il a été complètement présenté dans le chapitre 6. Le composant dépend du composant "DeviceInfoManager".

— **"MediaManager"** : ce composant permet de télécharger des fichiers à partir du web. Ces fichiers peuvent être des images, des musiques, des vidéos et des pdfs. Le composant dépend du composant "DeviceInfoManager".

— **"GoogleAPIParser"** : ce composant gère le parsing des réponses JSON ou XML retournés par les Web services de googles (Directions API et Places API). Il s'occupe plus particulièrement de transformer toutes les données en structures exploitables dans les codes sources Android et iOS de l'application. Il fonctionne en mode synchrone. Il contient quand même des delegates pour la gestion des erreurs. En effet, les web services de google peuvent retournés des erreurs tels que "requête non valide" ou encore "timeout", etc. Chaque erreur est gérée par une méthode du délégate.

Ces composants sont très différents et mettent tous en avant un aspect de nos composants multiplateformes. Ainsi, certains composants ont des méthodes de configuration. Certains fonctionnent en mode asynchrone, d'autres en mode synchrone. Et bien sûr, certains ont des délégates alors que d'autres non. Avec ces cinq composants nous avons donc intégré tous les types de traitements possibles.

Après avoir implémenté ces composants, nous les avons intégré dans le code source de l'application LocaPlace sur Android et iOS, figure 9.2. Pour ce faire, nous avons utilisé le logiciel présenté dans le chapitre 8, section 8.3.2. Même si c'est nous qui avons conçu cette solution, il était indispensable d'utiliser le logiciel. Nous aurions perdu trop de temps à rechercher les informations nécessaires à l'intégration dans l'interface publique et complémentaire de chaque composant. En terme d'intégration, nous avons principale invoquer nos composants à partir des contrôleurs de l'application LocaPlace.

Pour résumer, l'application LocaPlace a été réalisée pour Android et iOS. Pour ce faire, nous avons implémenté la structure de l'application en langage natif sur Android et iOS dans deux projets séparés. Ensuite, nous avons intégré les cinq composants multiplateformes présentés dans l'application. Bien sûr, ces deux étapes ne sont pas séparées. Elles se font de façon complémentaires pendant la phase de développement.

9.4 Évaluations

Pour l'évaluation de notre solution, nous comparons la version implémentée avec COMMON avec les versions 100% native. Ainsi, nous avons vérifié que notre solution ne bride pas les fonctionnalités offertes par le développement mobile natif. Ensuite, nous validons que

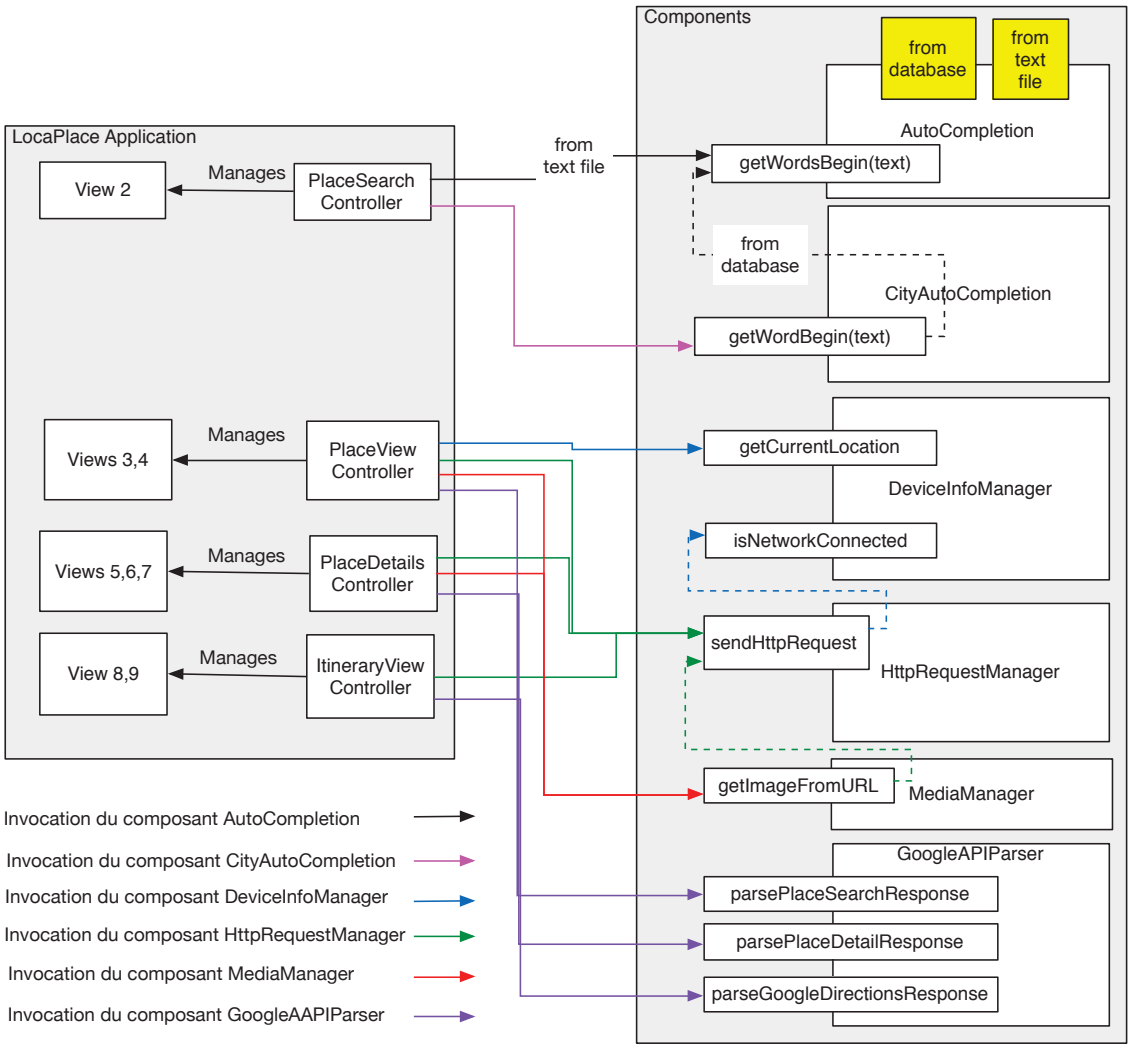


FIGURE 9.2 – Intégration des composants dans l'application LocaPlace

notre solution apporte un gain en temps de développement par rapport au développement 100% natif pour Android et iOS. Pour rappel, c'est notre principal objectif. Nous voulons cacher les différences entre plates-formes mobiles et ainsi diminuer la charge de développement d'applications multiplateformes. Enfin, nous vérifions que les applications générées sont aussi performantes que des applications natives.

9.4.1 Faisabilité de l'application

Avant de commencer l'étude des performances de l'application, il est nécessaire de discuter de la faisabilité de l'application. En effet, l'application LocaPlace adopte une interface utilisateur classique pour ce genre d'application, sans pour autant négliger les interactions possibles. Par exemple, lorsque l'utilisateur de l'application clique sur un lieu dans la liste des points d'intérêts, il arrive sur une vue avec le détail du point d'intérêt sélectionné. Il peut alors passer avec une action de swipe de son doigt au point d'intérêt précédent ou suivant, figure 9.3. Cette interaction est devenue commune sur mobile. Il est donc nécessaire que nous puissions la prendre en compte avec notre solution.

Comme, nous l'avons montré dans la section 9.3, l'interface utilisateur est complètement

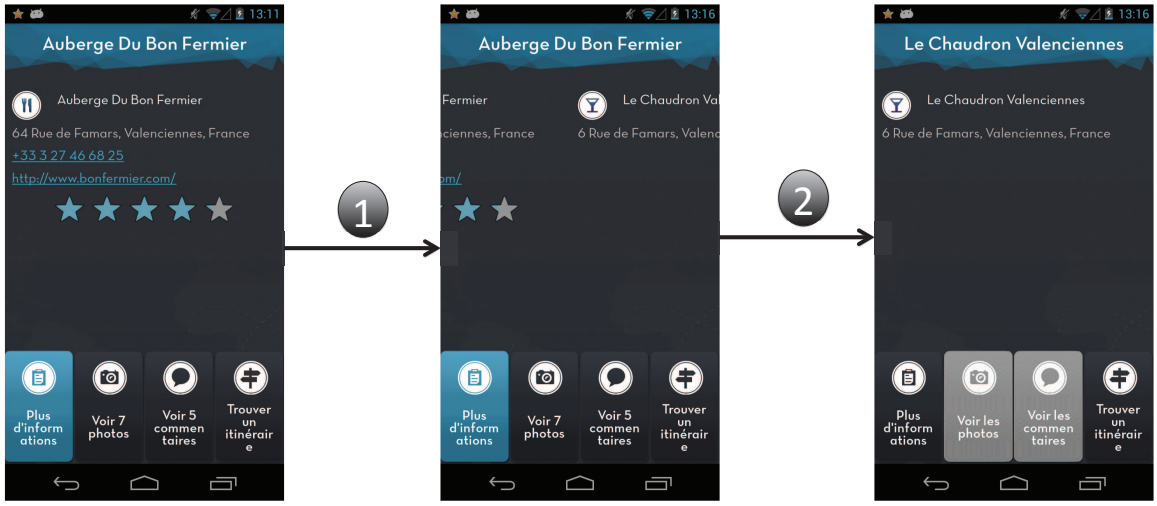


FIGURE 9.3 – Interaction de type swipe dans l'application LocaPlace

implémentée en langage natif. Plus généralement, tout ce qui n'est pas possible de faire avec nos composants peut être implémenté en langage natif. Par conséquent, notre solution ne limite pas le développement d'applications mobiles. Si une fonctionnalité est disponible sur une plate-forme mobile, l'utilisateur de notre solution pourra l'implémenter. Ceci est possible grâce à la complémentarité de notre solution et plus particulièrement de notre langage commun avec les environnements de développements natifs de chaque plate-forme cible. Il a donc été possible pour nous d'implémenter toute l'application avec les mêmes fonctionnalités que l'on aurait pu avoir avec du développement 100% natif. De plus, l'application garde l'intégrité de chacune des plates-formes cibles : Android et iOS.

9.4.2 Diminution de la charge de développement

En implémentant l'application de façon 100% native, nous pouvons comparer le nombre de lignes de code gagnées ou perdues en développant la même application avec COMMON. Sur le tableau 9.1 [5], nous donnons le nombre de lignes de code implémentées pour la version 100% native et la version avec COMMON. Avec notre solution, nous gagnons 2748 lignes de code soit 29% sur Android et 3878 lignes soit 35% sur iOS. Avec notre solution, nous gagnons donc au total plus de 6600 lignes de code sur les deux versions soit 32% du code total pour cette application multiplateforme. En cachant les différences entre plates-formes cibles pendant l'intégration de nos composants, nous gagnons donc une charge de travail non négligeable pour les développeurs mobiles.

Actuellement, nous ne sommes pas capables de chiffrer le temps gagné de façon exacte car le développement des deux applications a été effectué par le même développeur. Pour comparer le temps de développement, il aurait fallu effectuer la même application avec deux développeurs de même niveau. Cependant, à partir de notre expérience personnelle dans l'utilisation de COMMON, nous pouvons dire que l'application implémentée avec COMMON permet de gagner entre 30% et 50% de temps. En effet, les fonctionnalités proposées par nos composants n'ont pas besoin d'être testées par l'intégrateur. Par conséquent, lors du développement de son application, en plus de gagner des lignes de code, il va gagner du temps sur les tests de son application. Par exemple, il n'aura pas besoin de vérifier que le parsing des

5. Les chiffres donnés dans le tableau ne prennent pas en compte la librairie de scanner utilisée pour les deux applications.

Table 9.1 – Nombre de lignes de code gagnées avec COMMON

Version de LocaPlace	Nombre de lignes de code sur Android	Nombre de lignes de code sur iOS
100% native	9499	10954
Implémenté avec COMMON	6751	7076
Nombre de lignes de code gagnées	**2748** (**29%**)	**3878** (**35%**)

réponses JSON des web services de google est correct. Cette action aura déjà été effectuée pendant le développement du composant "GoogleAPIParser" par le développeur du composant. Dans nos futurs travaux, il serait intéressant de quantifier le temps de développement. Nous sommes persuadés que nous pouvons atteindre jusqu'à 50% de temps gagné lors de la phase de développement.

Le nombre de lignes de code pour chacune des versions a été récupéré à partir de la ligne de commande présentée dans le code 9.1 pour les projets Android et 9.2 pour les projets iOS. Cette commande permet de récupérer tous les fichiers ayant les extensions souhaitées (.java pour Android, .m, .mm etc. pour iOS) et ensuite pour chaque fichier de récupérer le nombre de lignes. Cette commande est moins précise que l'utilisation de Metrics sous eclipse utilisé dans le chapitre précédent, section 8.5 mais elle est utilisable pour les projets Android et iOS.

Code 9.1 – Commande pour calculer le nombre de ligne de code d'un projet android

```
1 find . \( -iname \*.java \) -exec wc -l '{}' \+
```

Code 9.2 – Commande pour calculer le nombre de ligne de code d'un projet iOS

```
1 find . \( -iname \*.m -o -iname \*.mm -o -iname \*.c -o -iname \*.cc -o -
      iname \*.h \) -exec wc -l '{}' \+
```

Pour atteindre 32% de lignes de code gagnées, nous n'avons eu besoin que d'intégrer 5 composants. Cependant, il est difficile d'en intégrer plus dans cette application. En effet, la limite à l'intégration des composants sont toutes les fonctionnalités spécifiques à l'application. Plus l'application a des spécificités, plus le développeur devra implémenter du code natif. Nous considérons que sur cette application, environ 5000 lignes de code sont dédiées à des spécificités (navigation, IHM, etc.). Pour obtenir ce chiffre, nous avons déduit manuellement la partie IHM de l'application. Maintenant, si nous prenons en compte uniquement le reste du code et que nous comparons maintenant les deux versions, nous avons le tableau 9.2. Dans ce cas, sur le code non dédié à des spécificités, nous obtenons un résultat de 63% de lignes de code gagnées. En fin de compte, sur la partie indépendante de l'application, ce à quoi nos composants doivent répondre, nous gagnons plus de 60% de code, ce qui est assez considérable. Par la suite, nous pourrons étendre nos composants à la partie ergonomie.

Même si les utilisateurs de COMMON (les intégrateurs de nos composants) ne sont pas les développeurs de composants, il est intéressant d'étudier la charge de travail affectée aux composants. Sur le tableau 9.3, nous récapitulons le nombre de lignes de code implémentées pour chaque composant sur Android et iOS. Nous remarquons que la charge totale est de 16000 lignes de codes alors qu'en utilisant nos composants dans l'application LocaPlace nous avons "seulement" gagné 6600 lignes. Cette différence s'explique par le fait que nos compo-

Table 9.2 – Nombre de lignes de code gagnées avec COMMON sans prendre en compte les spécificités de l'application LocaPlace

Version de LocaPlace	Nombre de lignes de code sur Android	Nombre de lignes de code sur iOS
100% native	4499	5954
Implémenté avec COMMON	1751	2076
Nombre de lignes de code gagnés	**2748** (**61%**)	**3878** (**65%**)

sants sont réalisés de façon générique pour être utilisés dans plusieurs configurations. Par exemple, le composant d'autocomplétion peut être utilisé à partir d'un tableau, d'un fichier texte, d'un fichier CSV ou d'une base de données. Dans l'application LocaPlace, il est uniquement configuré pour faire des recherches à partir d'un tableau. Les composants sont donc naturellement plus modulables que si nous avions implémenté la même fonction dans une application particulière. Cette modularité à un coût qui s'exprime en lignes de code. Dans le même ordre d'idée, le composant "MediaManager" permet de télécharger des images, des musiques, des vidéos et des fichiers pdf alors que dans l'application LocaPlace, nous utilisons uniquement le téléchargement des images. Toutes les fonctionnalités des composants ne sont donc pas utilisées dans LocaPlace. Cela explique en partie le surplus de code.

Table 9.3 – Nombre de lignes de code gagnées avec COMMON

Composant	Nombre de lignes de code sur Android	Nombre de lignes de code sur iOS
AutoCompletion	1289	1074
CityAutoCompletion	421	261
DeviceInfoManager	516	829
HttpRequestManager	1455	906
MediaManager	2407	3761
GoogleAPIParser	1505	1559
Total	**7593**	**8390**

Actuellement à Keyneosoft, nous intégrons les composants que nous développons. Pour le moment, ils sont intégrés nativement dans nos applications. Par la suite, il le seront avec notre langage commun. Il est indispensable pour nous de rentabiliser les composants, c'est-à-dire que le nombre de lignes gagnées soit supérieur au nombre de lignes de nos composants. Sans ça notre solution n'est pas viable dans l'entreprise. Pour cela, nous misons sur leur réutilisation. C'est une des propriétés essentielles de nos composants multiplateformes. Nous devons donc faire très attention à ne pas développer des composants qui sont spécifiques à une seule application. Dans ce cas, le surplus de code lié à la modularité du composant ne sera jamais rentabilisé. Aujourd'hui, plusieurs composants existants ont déjà été rentabilisés. Par exemple, le composant "HttpRequestManager" ainsi que le composant "DeviceInfoManager" ont déjà été intégrés plus d'une dizaine de fois dans nos applications. Dans le cas de

l'application LocaPlace, il suffirait d'implémenter trois fois le même style d'application pour commencer à rentabiliser les composants présentés ici.

9.4.3 Performances

Après avoir montré que notre solution nous permet de gagner du temps lors du développement d'une application multiplateforme, nous devons vérifier que l'application générée fournit des performances acceptables pour le mobile. Pour rappel, notre objectif est de fournir des applications mobiles avec des performances égales au natif. Pour ce faire, nous avons comparé les processus fournis par nos composants fonctionnant dans l'application LocaPlace avec les mêmes processus fonctionnant dans la version 100% native sur Android et iOS.

Relevé des valeurs

Pour chacun des processus testé, nous récupérons le temps moyen d'exécution. Chaque relevé consiste à récupérer la date courante sur le smartphone avant le départ du processus. Ensuite, à la fin du processus, nous récupérons une nouvelle fois la date courante sur le smartphone. Le temps d'exécution est alors calculé en faisant la date de fin soustrait à la date de début du processus. Nous obtenons alors le temps d'exécution du processus. Chaque temps relevé est alors enregistré dans un fichier CSV sur le smartphone. Nous effectuons pour chaque processus le traitement 100 fois et nous calculons la moyenne de tous les temps d'exécution relevés. Nous obtenons finalement le temps moyen d'exécution de chacun des processus.

Les processus testés sont :

— **L'autocomplétion d'un mot à partir d'un fichier texte** (vue 2) : ce processus est effectué au moment de la sélection par l'utilisateur du type des points d'intérêts recherchés.

— **L'autocomplétion d'un mot à partir d'une base de données** (vue 2) : ce processus est effectué au moment de la sélection par l'utilisateur de la ville dans laquelle il recherche des points d'intérêts.

— **La récupération des données de localisation du smartphone** (vue 3) : ce processus est effectué au moment de la recherche de points d'intérêts. Il consiste au lancement du service de localisation jusqu'au retour du service avec la localisation du smartphone.

— **L'envoi d'une requête à un web service HTTP** (vues 3, 5, 6, 8 et 12) : ce processus est effectué sur pratiquement toutes les pages de l'application. Il consiste à la création d'un client HTTP et l'envoi de la requête. La réception de la réponse du web service n'est pas prise en compte vu qu'elle dépend fortement de la disponibilité du serveur et du débit internet possible avec le smartphone au moment de l'appel.

— **Parsing des réponses du web service Place Search** (vue 3) : ce processus est effectué lors de la recherche des points d'intérêts de l'utilisateur. Il consiste au parsing de la réponse sous format XML. Pour que le test soit concluant, nous avons effectué le parsing sur la même recherche 100 fois. Ainsi les données retournées par le serveur étaient toujours les mêmes.

— **Parsing des réponses du web service Place Details** (vue 5) : ce processus est effectué lors de la demande de plus d'informations par l'utilisateur sur un point d'intérêt particulier. Il consiste au parsing de la réponse sous format XML. Pour que le test soit concluant, nous avons effectué le parsing sur le même lieu 100 fois. Ainsi les données retournées par le serveur étaient toujours les mêmes.

— **Parsing des réponses du web service Google Directions** (vues 8 et 12) : ce processus est effectué lors de la recherche d'un itinéraire par l'utilisateur vers un point d'intérêt particulier. Il consiste au parsing de la réponse sous format XML. Pour que le test soit concluant, nous avons effectué le parsing sur la même recherche 100 fois. Ainsi les données retournées par le serveur étaient toujours les mêmes.
— **La formation d'un itinéraire** (vues 8 et 12) : ce processus est effectué après la demande par l'utilisateur d'un itinéraire pour aller jusqu'à un point d'intérêt particulier. Il consiste uniquement à la création de la structure de données contenant tous les points de l'itinéraire et pouvant être affichée sur une carte.

Les relevés des temps moyens d'exécution de chaque processus ont été effectués sur trois smartphones différents pour chaque OS. Sur Android, nous avons effectué nos relevés sur :
— **Samsung Google Galaxy nexus i9250** avec Android 4.2.1. Ce smartphone fabriqué par Samsung est équipé d'un processeur TI OMAP 4460 Dual-core 1,2 GHz Cortex A9 et d'1GB de RAM. Il est sorti en novembre 2011.
— **Samsung galaxy note 3** avec Android version 4.3. Ce smartphone fabriqué par Samsung est équipé d'un processeur Exynos 5 Octa 5420 Quad-core 1,3 GHz Cortex A7 et de 3GB de RAM. Il est sorti en septembre 2013.
— **Nexus 5** avec Android version 4.4.2. Ce smartphone fabriqué par LG est équipé d'un processeur Qualcomm MSM8974 Snapdragon 800 Quad-core 2,3 GHz Krait 400 et de 2GB de RAM. Il est sorti en novembre 2013.

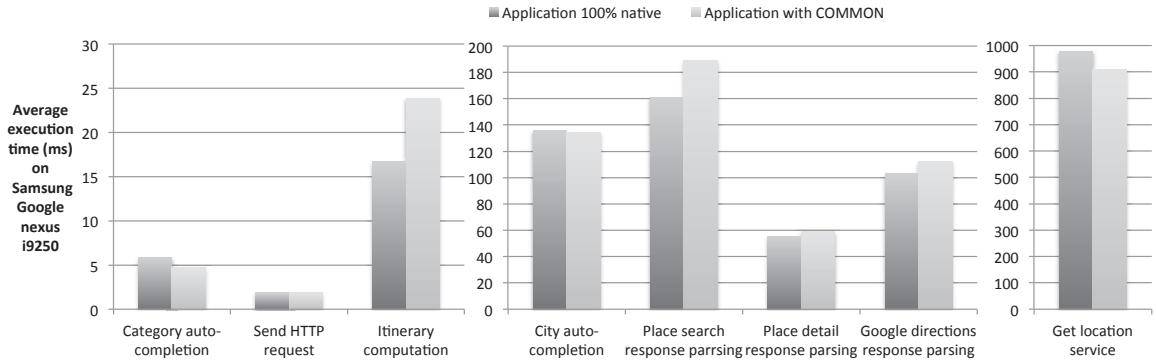
Sur iOS, nous avons effectué nos relevé sur :
— **iPhone 4** avec iOS version 7.0.2. Ce smartphone est équipé d'un processeur Apple A4 1 GHz Cortex-A8 et de 512MB de RAM. Il est sorti en juin 2010.
— **iPad mini** avec iOS version 7.0.4. Cette tablette est équipée d'un processeur Apple A5 Dual-core 1 GHz Cortex-A9 et de 512MB de RAM. Il est sorti en novembre 2012.
— **iPhone 5s** avec iOS version 7.0.4. Ce smartphone est équipé d'un processeur Apple A7 Dual-core 1.3 GHz Cyclone et de 1GB de RAM. Il est sorti en septembre 2013.

Tous les smartphones iOS sont fabriqués par Apple.

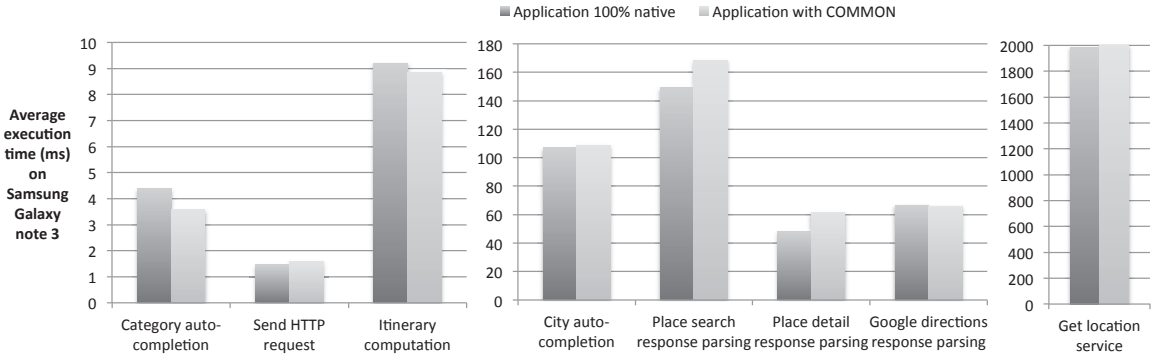
Résultats

Sur les figures 9.4a, 9.4b et 9.4c ainsi que sur les figures 9.5a, 9.5b et 9.5c, nous donnons les temps d'exécution moyen de chaque processus relevé respectivement sur Android et sur iOS à travers la version 100% native de l'application LocaPlace et la version implémentée avec COMMON. Sur les deux plates-formes, nous constatons que les deux versions de l'application fournissent des performances égales sur les différents smartphones choisis. Parfois l'application générée avec COMMON fournit des temps plus élevés mais cela est toujours négligeable, pas plus de 10% de temps d'exécution supplémentaire. Il arrive même parfois que l'application 100% native soit moins performante que l'application avec COMMON. Ces différences s'expliquent par le fait que le développement de l'application et des composants n'ont pas été effectués par le même développeur. Chacun a alors utilisé des structures de données différentes ce qui explique que les traitements sont plus ou moins rapides. Les développeurs ne se sont pas concertés avant le début de l'expérimentation. Quoi qu'il en soit les différences restent de toute façon négligeables.

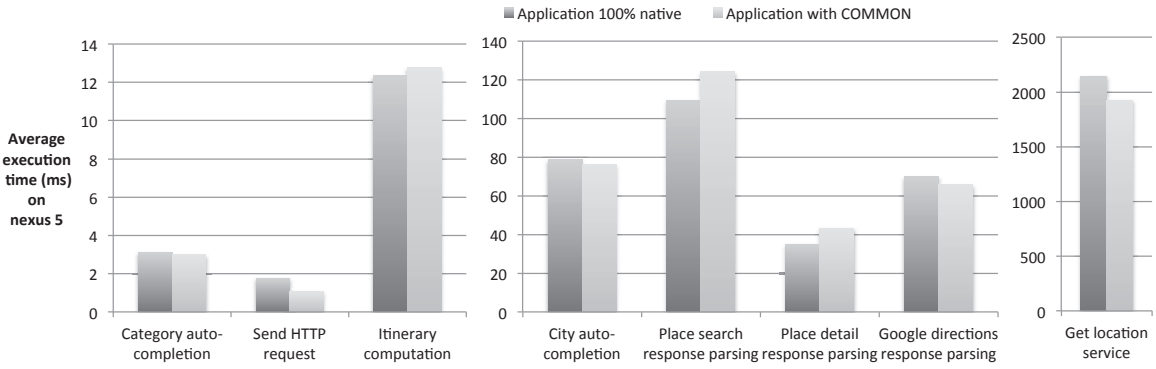
Pour récapituler, notre solution ne génère pas de surcoût à l'exécution. Nous fournissons les mêmes performances que les applications 100% natives. Ce résultat était prévisible. En effet, nous avions choisi d'implémenter nos composants multiplateformes de façon native sur chacune des plates-formes cibles en partie pour fournir des performances égales au natif. Cependant, notre crainte principale provenait plutôt des formats dans lesquels sont déployés nos



(a) Temps d'exécution moyen relevé sur un Samsung Google Galaxy Nexus i9250

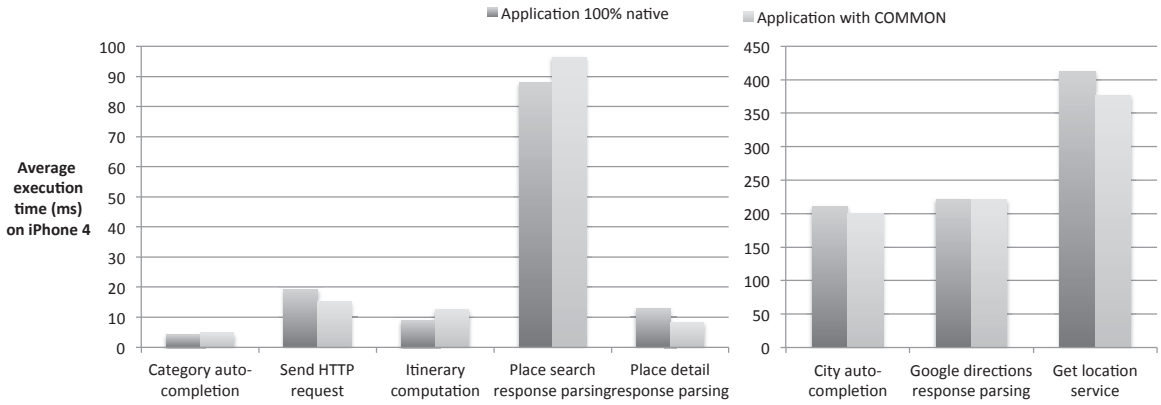


(b) Temps d'exécution moyen relevé sur un Samsung Galaxy note 3

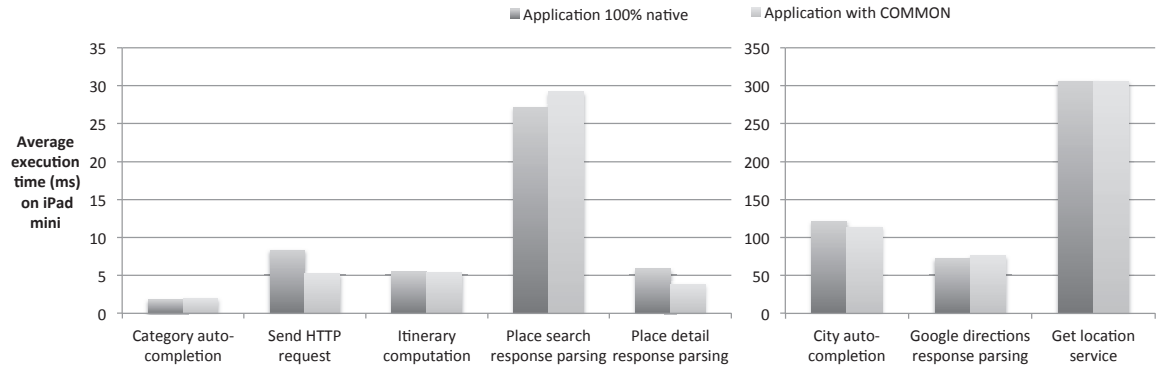


(c) Temps d'exécution moyen relevé sur un nexus 5

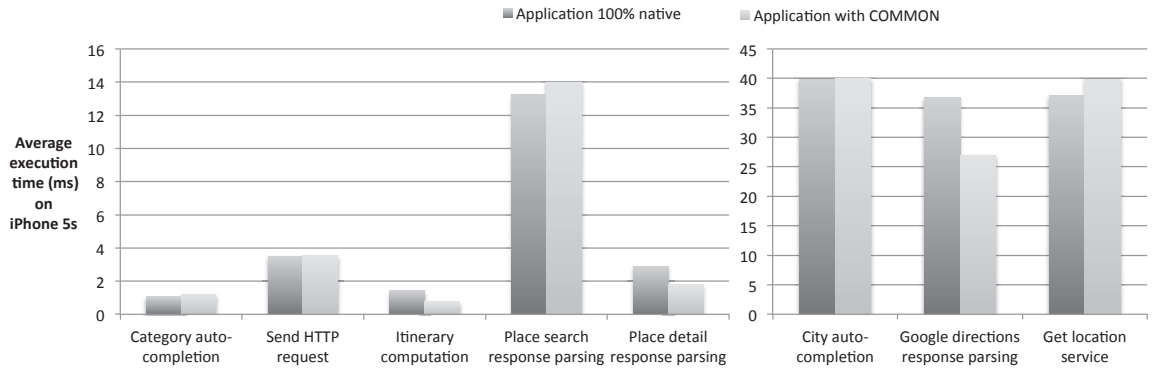
FIGURE 9.4 – Temps d'exécution moyen relevé sur plusieurs smartphones android



(a) Temps d'exécution moyen relevé sur un iPhone 4



(b) Temps d'exécution moyen relevé sur un iPad mini



(c) Temps d'exécution moyen relevé sur un iPhone 5s

FIGURE 9.5 – Temps d'exécution moyen relevé sur plusieurs smartphones iOS

composants. Sur Android, les implémentations des composants sont distribuées et intégrées sous format jar. Sur iOS, le format utilisé est le framework. Avec ces résultats, nous pouvons donc conclure que l'exécution d'un jar sur Android et d'un framework sur iOS et donc de nos composants ne génère pas de surcoût à l'exécution. De plus, le code généré par notre compilateur n'entraine pas non plus de surcoût à l'exécution.

Dans la suite, nous allons vérifier que le chargement des jars ou frameworks qui contiennent le code source des composants n'altère pas non plus la RAM des smartphones sur lesquels l'application fonctionne.

9.4.4 Consommation des ressources

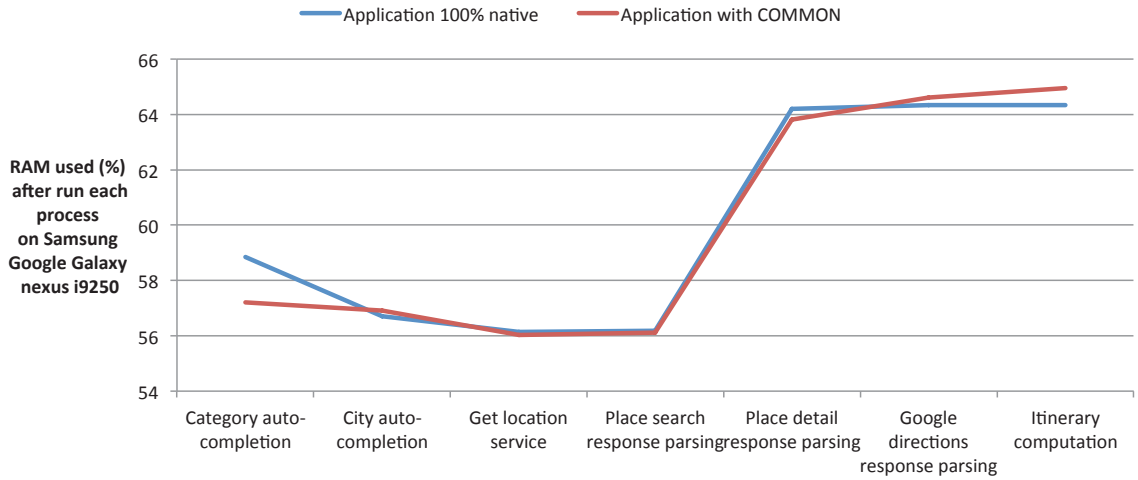
Utilisation de la RAM

Comme pour les temps d'exécution moyens calculés précédemment, nous avons effectué le même genre de relevés sur les mêmes processus mais avec comme objectif de récupérer le pourcentage de la RAM moyenne utilisée par chacun des processus. Nous voulons vérifier que le chargement en mémoire d'un composant (jar sur Android, framework sur iOS) pendant l'exécution ne sature pas la RAM du smartphone. Pour cela, nous relevons à la fin de chaque processus la RAM utilisée. Ainsi, nous pouvons comparer la RAM utilisée sur l'application 100% native avec celle de l'application utilisant COMMON. Comme précédemment, nous effectuons les relevés 100 fois et nous calculons ensuite la moyenne.

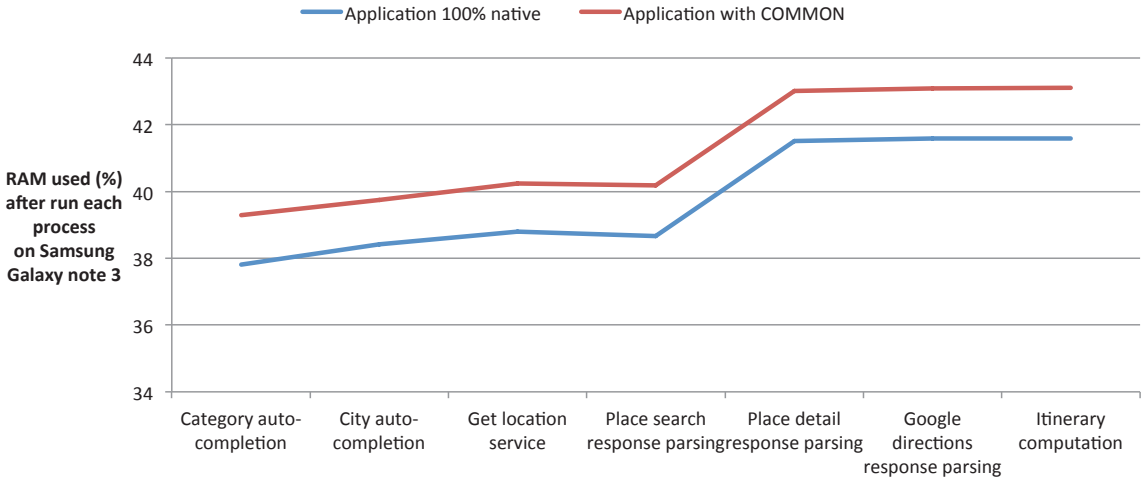
Sur les figures 9.6 et 9.7, nous donnons la consommation moyenne de RAM par processus sur plusieurs smartphones Android et iOS. Nous avons volontairement classé les processus par ordre d'apparition pendant l'utilisation de l'application LocaPlace. Ainsi, nous pouvons observer l'évolution de la consommation de la RAM en fonction du cycle d'utilisation de l'application. Nous n'avons pas monitoré le processus d'envoi de requête HTTP car il est utilisé à plusieurs moments du cycle de vie de l'application.

Globalement, que l'on soit sur Android ou iOS, nous constatons que, sur les deux plates-formes, la consommation de RAM se comporte de la même façon que l'on soit dans la version 100% native ou la version implémentée avec COMMON de LocaPlace. Cependant, nous devons quand même dissocier l'étude de la RAM sur Android et iOS. Effectivement, pour récupérer la valeur de la RAM utilisée, nous utilisons des APIs natives à chaque plate-forme. Malheureusement, ces APIs Android et iOS ne fonctionnent pas de la même façon. C'est une des conséquences de l'hétérogénéité dans le mobile. Sur Android, nous récupérons la RAM utilisée par tout le système alors que sur iOS, nous récupérons la RAM utilisée par l'application. Ainsi, sur iOS, nous sommes beaucoup plus précis alors que sur Android, nous ne pouvons pas voir le détail de la consommation spécifique à nos applications.

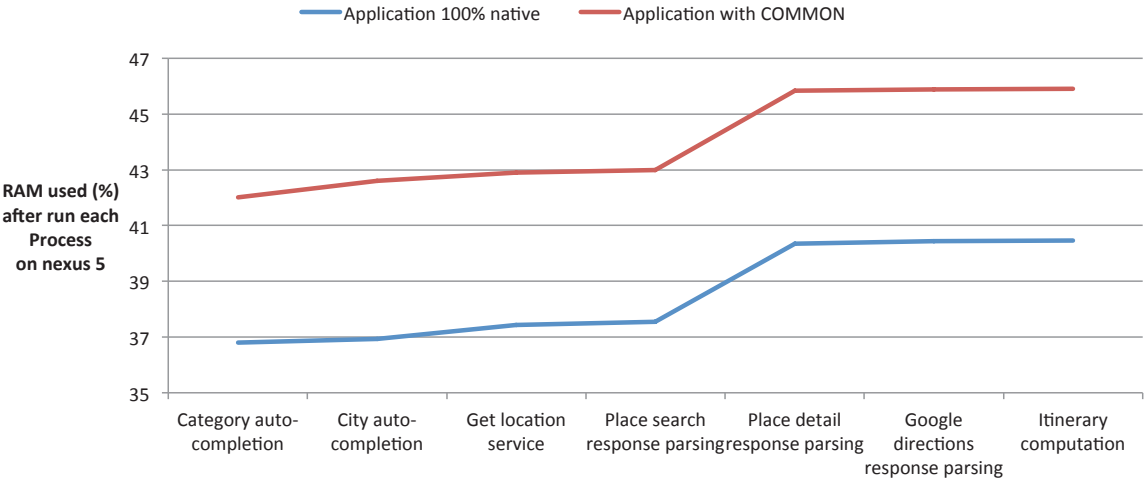
Sur Android, figures 9.6a, 9.6b, 9.6c nous constatons que les courbes d'utilisation de la RAM par chaque application ont les mêmes tendances. C'est d'ailleurs beaucoup plus visible sur les figures 9.6b et 9.6c. Cependant, la valeur de la RAM utilisée au lancement de l'application n'est jamais la même. Ceci s'explique par l'état du téléphone (connexion à internet activée, connexion bluetooth activée, réception d'un SMS etc.) au lancement de l'application. En fonction de cet état, il se peut que le système d'exploitation lance plus ou moins un grand nombre de routines en arrière plan qui utilisent chacune de la RAM. Même si nous avons essayé de limiter ces différences en désactivant au maximum les services Android, nous n'avons jamais réussi à atteindre le même niveau de RAM au lancement de l'application. Ce qui est important, ici, est que sur chaque graphique, les courbes ont les mêmes tendances. Cela signifie que la version 100% native et la version implémentée avec



(a) RAM moyenne utilisée après chaque processus sur un Samsung Google Galaxy Nexus i9250

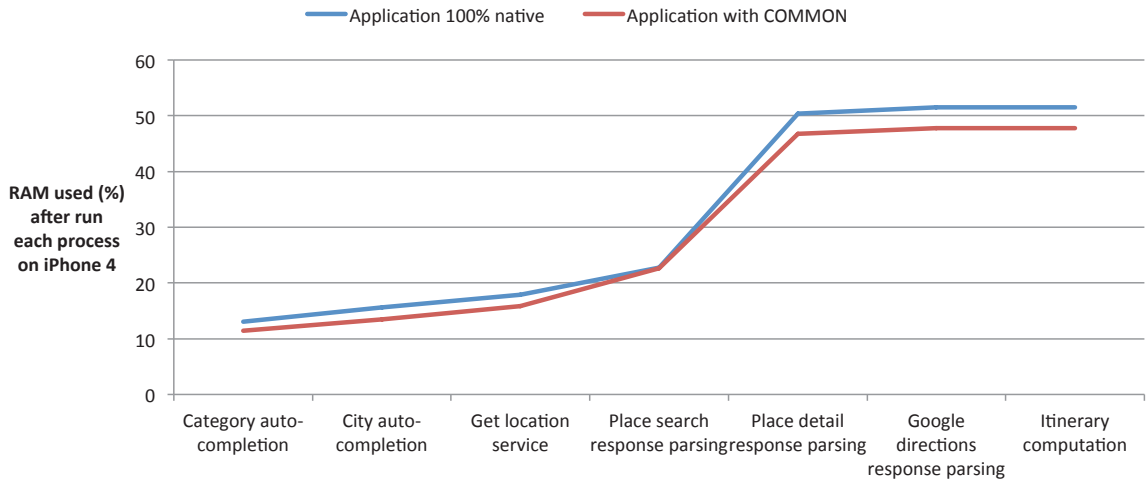


(b) RAM moyenne utilisée après chaque processus sur un Samsung Galaxy note 3

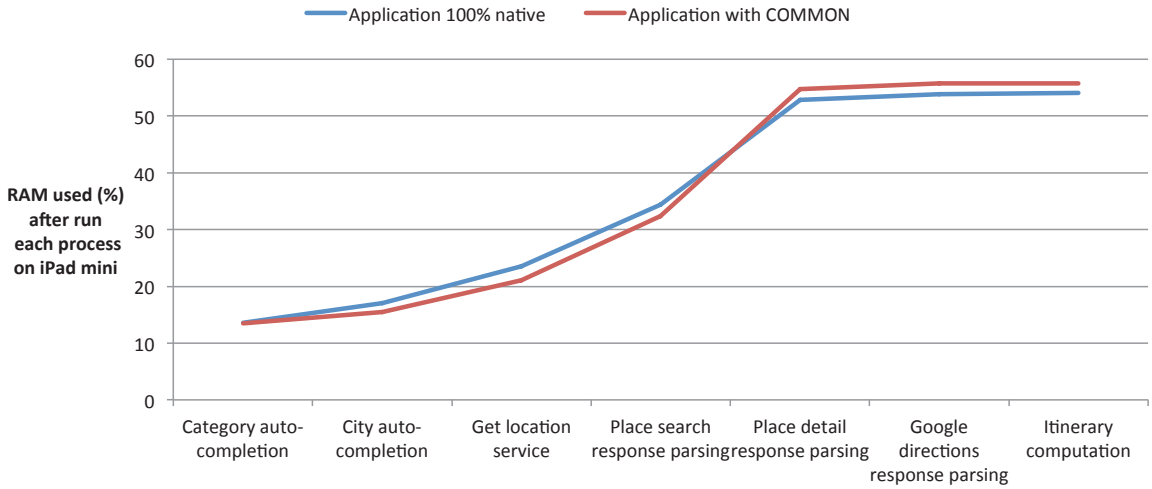


(c) RAM moyenne utilisée après chaque processus sur un nexus 5

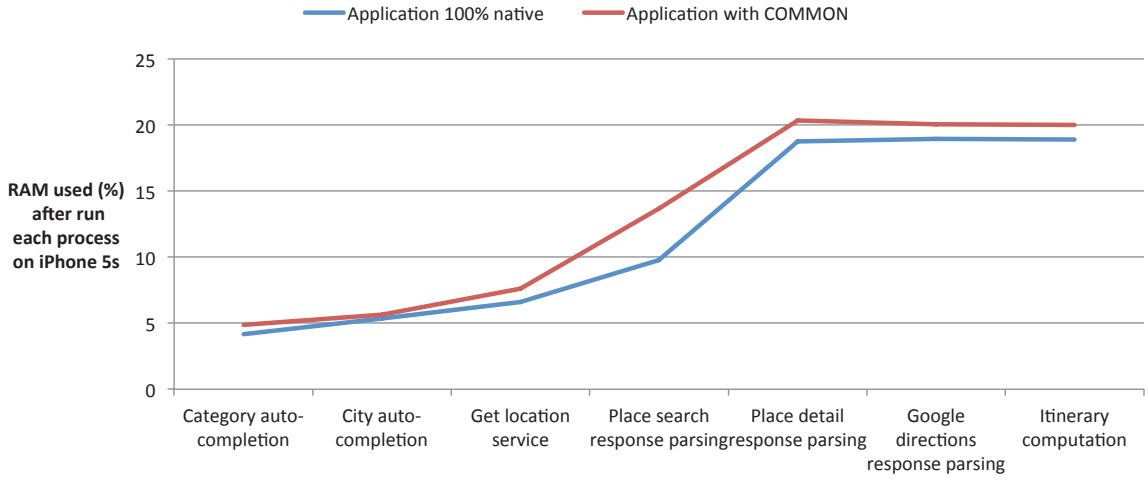
FIGURE 9.6 – RAM moyenne utilisée après chaque processus sur plusieurs smartphones android



(a) RAM moyenne utilisée après chaque processus sur un iPhone 4



(b) RAM moyenne utilisée après chaque processus sur un iPad mini



(c) RAM moyenne utilisée après chaque processus sur un iPhone 5s

FIGURE 9.7 – RAM moyenne utilisée après chaque processus sur plusieurs smartphones iOS

COMMON fonctionne de la même façon. Le chargement en mémoire des composants n'a donc pas d'impact sur l'utilisation de la RAM.

Sur iOS, figures 9.7a, 9.7b et 9.7c nous constatons le même résultat avec une précision beaucoup plus grande (à l'octet près). Nous n'observons pas de pics lors de l'utilisation de nos composants. Nous pouvons en conclure que l'utilisation d'un composant et par la même occasion son chargement en mémoire n'entraine pas de consommation supplémentaire de la RAM.

Poids de l'application

Nous terminons l'évaluation par la récupération du poids de chaque application sur chaque OS. L'objectif est de vérifier que les applications implémentées avec COMMON ne prennent pas plus de place que la même application 100% native dans la mémoire des stockages des smartphones. Les smartphones Android "low cost" sont souvent vendus avec 2 Go ou 4 Go de stockage de données. Il est donc primordial que les applications ne pèsent pas des dizaines de méga-octets (Mo). L'utilisateur serait alors bloqué rapidement dans l'utilisation de son smartphone. Dans le tableau 9.4, nous observons que pour chacune des versions de l'application sur Android, la différence de poids est négligeable et même égale. Sur iOs, tableau 9.5, nous observons la même chose. L'application 100% native packagée pour fonctionner sur iOS a le même poids que la version implémentée avec COMMON.

Table 9.4 – Poids de l'application LocaPlace installées sur plusieurs smartphones Android

Smartphones	Poids de la version 100% native	Poids de la version implémentée avec COMMON
Samsung Google Galaxy nexus i9250	9,43 Mo	9,43 Mo
Samsung galaxy note 3	9,7 Mo	9,7 Mo
nexus 5	7,8 Mo	7,8 Mo

Table 9.5 – Poids de l'application LocaPlace lors de son packaging pour être exécutée sur des smartphones iOS

	Poids de la version 100% native	Poids de la version implémentée avec COMMON
Application packagée	4,8 Mo	4,9 Mo

Nous pouvons en conclure que sur Android et iOS, l'utilisation de COMMON n'entraine pas de consommation supplémentaire des ressources du smartphone que ce soit en matière de RAM utilisée ou d'espace mémoire par rapport à la même application développée nativement. Ceci s'explique principalement par le fait que nous n'importons pas sur le smarpthone de modules supplémentaires liés à l'utilisation de COMMON.

9.5 Conclusion

Dans ce chapitre, nous avons effectué une évaluation de notre solution. Pour ce faire, nous nous sommes mis dans les conditions réelles de développement d'une application mobile. Nous avons choisi de développer l'application LocaPlace, une application de recherche de points d'intérêts, pour Android et iOS avec COMMON. Nous avons intégré cinq composants ayant des configurations et fonctionnement différents. Pendant la phase de développement des composants, nous avons utilisé notre logiciel de génération automatique des interfaces pour générer chacune des interfaces publiques et complémentaires. Bien sûr, pour intégrer nos composants, nous avons utilisé notre langage commun. De plus, cette intégration nous a permis de montrer que les concepts de méthodes de configuration, d'entrée et de sortie sont suffisamment flexibles pour que nos composants soient intégrés dans n'importe quelle configuration.

Après avoir développé l'application entière avec succès, nous avons comparé ses performances et sa consommation de ressources avec la même application implémentée nativement. Tout d'abord, nous avons montré que les versions de LocaPlace avec COMMON pour Android et iOS contiennent globalement 32% de code en moins que les versions natives. Cela représente une charge de travail en moins pour les développeurs d'applications mobiles considérable. Ensuite en matière de performances, nous avons montré que COMMON fournit des applications aussi performantes que nativement. L'utilisation de nos composants n'entraine donc pas de temps d'exécution supplémentaire. Nous avons observé la même tendance avec la consommation de ressources de l'application. Les applications implémentées avec COMMON consomment autant de ressources que celles natives.

Pour la première utilisation des composants dans LocaPlace, nous n'avons pas rentabilisé le temps passé à la réalisation du composant. Cependant, la programmation par composants est basée sur la réutilisation des composants. Par conséquent, à la deuxième utilisation des composants, nous rentabilisons l'effort effectué pendant la réalisation. À la troisième et plus, nous commençons à réellement gagner du temps pour la réalisation de nos applications. Comme nous l'avons souligné certains composants ont déjà été intégrés plus d'une dizaine de fois.

Avec cette évaluation, nous avons montré que notre solution répond correctement à nos besoins. Effectivement, COMMON permet de diminuer le temps de développement tout en gardant des performances identiques au développement natif. De plus, COMMON ne limite pas le développement mobile. Nous avons pu réaliser une application ayant un aspect natif et respectant les contraintes de chaque plate-forme. Dans la suite, nous allons la comparer avec plusieurs solutions disponibles sur le marché. C'est l'objet du prochain chapitre.

Chapitre 10

Comparaison de COMMON avec d'autres solutions

Dans ce chapitre, nous comparons COMMON avec trois produits : Phonegap, Titanium mobile et Xamarin. Ces trois produits reposent sur des approches différentes, ce qui nous permet de positionner COMMON par rapport à ces approches. Pour ce faire, nous avons débuté l'implémentation de LocaPlace avec ces trois technologies. Nous montrons ensuite que COMMON fournit les meilleures performances en matière de temps d'exécution et consommation de ressources. Dans la suite de nos travaux, nous terminerons totalement l'implémentation de LocaPlace avec ces trois technologies. Notre objectif est de comparer les gains en matière de nombre de lignes de code écrites par les développeurs mobiles.

Sommaire

10.1 Introduction 153
10.2 Présentation des outils 154
10.2.1 PhoneGap 154
10.2.2 Titanium mobile 155
10.2.3 Xamarin 157
10.3 Résultats 158
10.3.1 Expérience utilisateur 159
10.3.2 Performances 162
10.3.3 Utilisation des ressources 165
10.3.4 Discussions 168
10.4 Conclusion 169

10.1 Introduction

Dans ce chapitre, nous comparons notre framework de développement COMMON avec d'autres solutions disponibles sur le marché. Nous avons choisi PhoneGap, Titanium mobile et Xamarin. Ce sont trois technologies matures qui permettent de développer de façon multiplateforme une application mobile pour au minimum Android et iOS.

Ces trois solutions sont complètement différentes ce qui nous permet de comparer COMMON avec d'autres approches. PhoneGap permet d'embarquer un site web mobile dans une application mobile. Ce site web a bien sûr accès aux couches matérielles du smartphone. Titanium mobile permet d'écrire une application en Javascript et XML qui est ensuite interprétée en temps réel à l'exécution. Ces deux technologies sont donc basées sur un moteur d'exécution qui s'occupe de faire le lien en temps réel entre le site web ou l'application et le système d'exploitation. Quant à elle, Xamarin permet d'implémenter une application mobile en C# qui est ensuite complètement transformée en code natif. L'application déployée est donc 100% native comme les applications générées avec COMMON. À elles 4, COMMON, PhoneGap, Titanium mobile et Xamarin couvrent un panel assez large d'approches possibles dans le développement mobile multiplateforme.

Dans [HHM12a], les auteurs ont introduit plusieurs critères de comparaison de solutions de développement mobile multiplateforme. Ces critères se présentent en deux plans : le rendu final des applications générées et les outils de développement. Au niveau du rendu final des applications générées, les solutions doivent :

— Permettre d'accéder aux fonctionnalités spécifiques de chacune des plates-formes cibles.
— Fournir un aspect natif tout en gardant l'intégrité de chaque plate-forme cible.
— Fournir un comportement fluide et rapide.

Au niveau du développement, les solutions doivent :

— Fournir un environnement de développement avec des outils de débogage, des émulateurs etc.
— Faciliter le développement grâce à une documentation claire et accessible rapidement.
— Faciliter la maintenance des applications.
— Accélérer le coût de développement de l'application.

Tout au long de ce chapitre, nous utilisons tous ces critères pour notre comparaison. Dans l'article [HHM12a], ces critères ont servi à comparer PhoneGap et Titanium mobile.

La première partie de ce chapitre, section 10.2, est consacrée à la présentation de PhoneGap, Titanium mobile et Xamarin. Dans la section 10.3, nous comparons les résultats obtenus avec chacune des technologies. Nous finissons par une conclusion, section 10.4.

10.2 Présentation des outils

Dans cette section, nous nous focalisons sur l'architecture de l'application LocaPlace implémentée avec chacune des solutions. À travers ces architectures, nous présentons ainsi les spécificités de chacune d'entre elles.

10.2.1 PhoneGap

L'architecture de l'application LocaPlace implémentée avec PhoneGap est présentée sur la figure 10.1. En matière de développement, elle doit être vue comme un site web et non comme une application mobile. D'ailleurs LocaPlace avec PhoneGap pourrait très bien fonctionner à travers un navigateur internet. Elle est divisée en deux parties :

— **Interface utilisateur** : elle est développée en HTML et CSS, pour la partie affichage, ainsi qu'en Javascript, pour la récupération des interactions utilisateurs. Dans le but de fournir un aspect de type application mobile, ce site web a été implémenté en suivant les règles du responsive design [Fra12]. Elle s'adapte à plusieurs types d'écrans (smartphones, tablettes). Nous utilisons aussi des plugins tels que JQuery mobile ou

Sencha touch qui sont des frameworks Javascript développés pour gérer les interfaces mobiles. Même si la plupart du code est identique dans cette partie, il est possible de diviser le code source en fonction de la plate-forme cible. Par exemple, dans l'application LocaPlace, sur iOS, il faut afficher un bouton de retour dans la barre de navigation alors que sur Android, ce bouton n'existe pas. Dans ce cas, nous devons diviser le code source en deux fichiers distincts.

— **Services** : ils représentent les fonctionnalités que nous offrons à travers nos composants. Ils se basent sur les APIs PhoneGap pour fonctionner. Ces APIs sont communes à toutes les plates-formes cibles et sont accessibles uniquement en Javascript. Elles font le lien entre le code source de l'application et les fonctionnalités des plates-formes cibles. De plus, certains services peuvent utiliser des plugins supplémentaires.

Même si l'application LocaPlace avec PhoneGap ressemble un site web, PhoneGap n'offre pas les fonctionnalités telles que la mise à jour du code source dynamiquement offerte habituellement par les serveurs d'applications web.

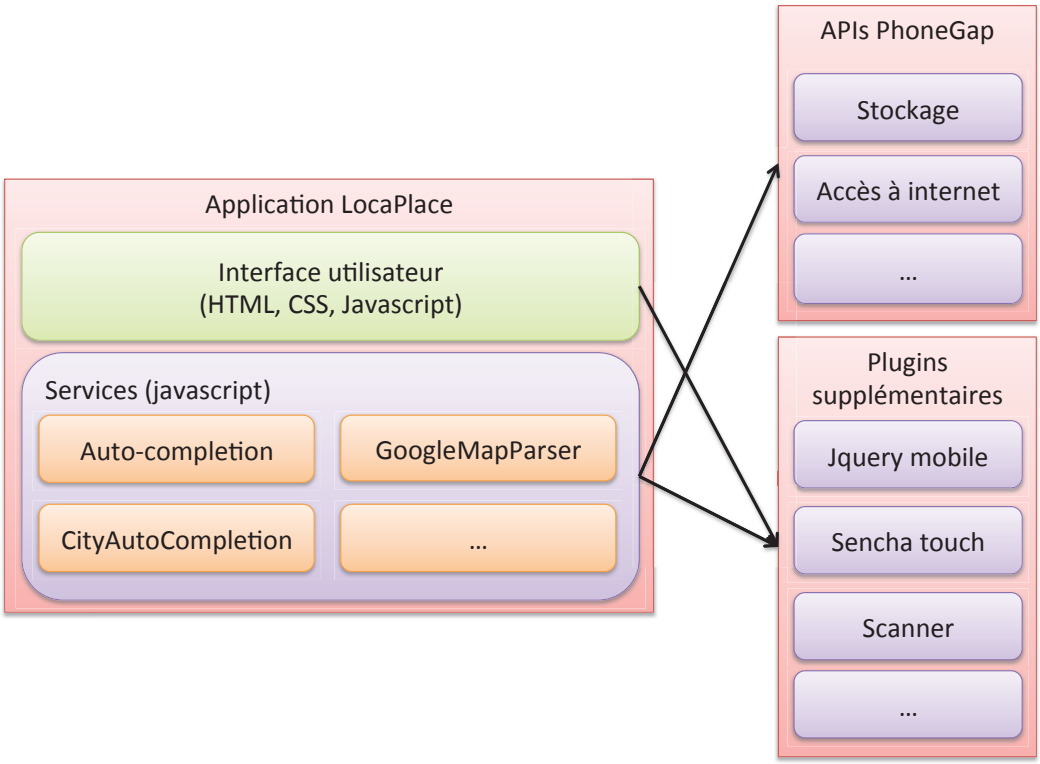


FIGURE 10.1 – Architecture de l'application LocaPlace implémentée avec PhoneGap pour Android et iOS

Le code source écrit à l'aide de technologies web est ensuite encapsulé dans une application native qui contient un moteur d'interprétation du code source PhoneGap et des plugins compatibles. À l'exécution, tout le code est donc interprété en temps réel.

10.2.2 Titanium mobile

L'architecture de l'application LocaPlace implémentée avec Titanium mobile est présentée sur la figure 10.2. Elle est divisée en trois parties distinctes :

— **L'interface utilisateur** : elle est développée en XML et Javascript avec le framework Alloy. Alloy permet de créer la partie graphique de l'application en XML. Nous pouvons apparenter ce processus à ce qui est fait sur Android avec les layouts. La

gestion des évènements (clic sur un bouton, etc.) est, quant à elle, implémentée dans un contrôleur de vues en Javascript.

— **Services** : ils sont développés entièrement en Javascript. Ils correspondent aux tâches de nos composants multiplateformes. Pour effectuer leurs processus, ils se basent sur les APIs disponibles dans le SDK Titanium. Les APIs de ce SDK représentent toutes les fonctionnalités que l'on peut trouver sur le mobile telles que la géolocalisation, l'envoi de requête HTTP, l'accès aux bases de données, etc.

— **Modules** : ils peuvent être considérés comme des services qui sont, cette fois, développés nativement (sans passer par Titanium mobile). Titanium mobile s'occupe ensuite de créer les proxys pour intégrér les mobiles à partir d'un code Javascript. Dans l'application, nous utilisons deux modules : le scanner fourni par l'entreprise Scandit [1], et l'API de map fournit par google (affichage de cartes, itinéraires, etc.). Nous aurions aussi pu intégrer notre composant de scanner. Les modules sont téléchargeables à partir d'un marketplace.

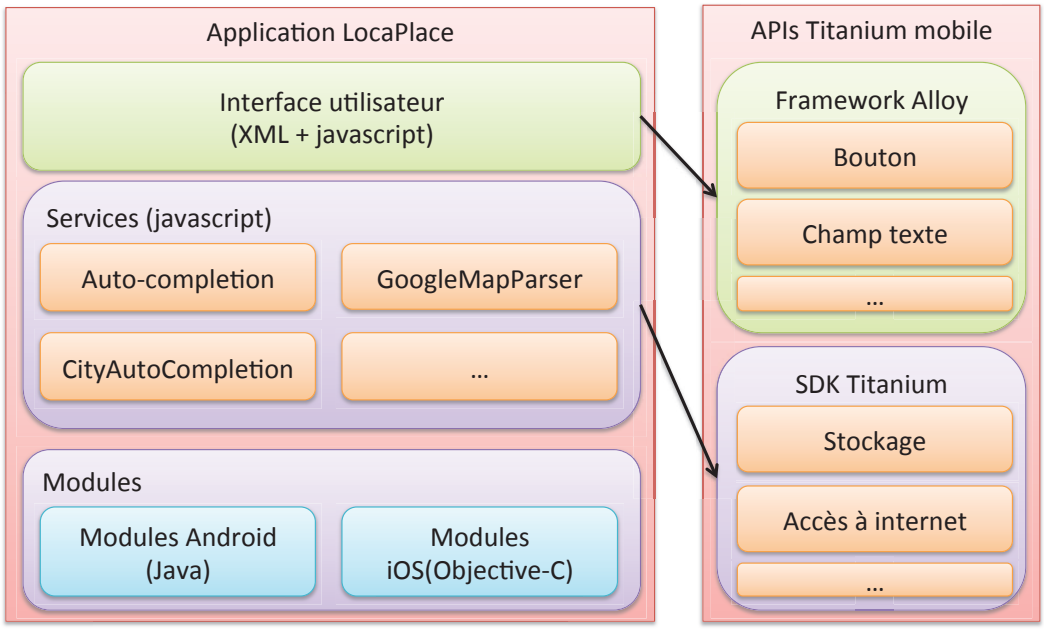


FIGURE 10.2 – Architecture de l'application LocaPlace implémentée avec Titanium mobile pour Android et iOS

Même si sur la figure 10.2 tout le code de l'application est commun à Android et iOS, en réalité, le code source écrit avec Titanium mobile peut être différencié en fonction d'une plate-forme mobile. Par exemple, lors de la définition de l'interface utilisateur, nous avons en fonction d'Android et d'iOS, divisé le code en deux parties. Ainsi, nous avons développé l'interface de deux façons différentes dans un seul code source commun. De la même manière, dans le code lié aux services communs, parfois, nous avons divisé le code source en fonction de la plate-forme cible. Ces différenciations assez fréquentes du code s'expliquent par le fait que les APIs fournies par Alloy ou Titanium n'ont pas obligatoirement le même comportement en fonction de la plate-forme cible.

En matière de distribution, tout le code source de l'application est importé dans un package android exécutable sur les smartphones Android et un exécutable de type application iOS. Le code est accompagné d'un interpréteur qui, à l'exécution, l'interprète et affiche le résultat sur le smartphone. L'aspect final de l'application est très proche de l'aspect d'une

1. Site web de Scandit : http ://www.scandit.com

application native.

10.2.3 Xamarin

Avant de détailler l'architecture de l'application implémentée avec Xamarin, figure 10.3, il est intéressant d'étudier l'approche suivie par Xamarin. Le SDK Xamarin est composé de trois frameworks :

- **Mono.Android.dll** : c'est une représentation du SDK android en C#. Toutes les classes, méthodes du SDK Android sont représentées. Seul le langage change. Toutes les spécificités Android telles que les fragments, les notifications locales à l'aide de broadcast, le partage de données entre applications, etc., elles, ne changent pas du tout. Ce n'est pas l'implémentation du SDK Android qui est transposé en C# mais uniquement les fonctionnalités de celui-ci (les interfaces). Nous pouvons voir le Mono.Android.dll comme une passerelle en C# qui permet d'accéder à l'implémentation Java du SDK Android. Le lien entre la représentation du SDK et l'implémentation se fait à la compilation.
- **Monotouch.dll** : c'est une représentation du SDK iOS en C#. Comme pour le SDK android, toutes les classes et méthodes d'iOS sont représentées sans aucune modification.
- **.Net framework** : ce framework contient des librairies communes à chacune des plates-formes cibles (accès à internet, au stockage, etc.). Le code écrit à partir de ce framework est donc réutilisable par chacune des applications. Nous imaginons que l'implémentation de ce framework est basée sur Mono.Android.dll et Monotouch.dll.

Avec cette composition, toutes les classes et méthodes de chacun des SDKs android et iOS sont accessibles. Il est donc possible d'accéder à toutes les spécificités de chacun des systèmes d'exploitation. Par exemple, sur Android, un développeur peut utiliser les fonctionnalités liées au NFC sur Android même si cela n'existe pas sur iOS. De la même manière, un développeur peut implémenter une interface utilisateur complètement liée à Android ou iOS. Il aura accès à tous les éléments natifs des SDK à travers Mono.Android.dll et Monotouch.dll. L'application a donc complètement l'aspect d'une interface développée nativement.

Contrairement aux solutions précédentes, avec Xamarin, l'application LocaPlace est divisée en deux versions : une pour android et une autre pour iOS, figure 10.3. Ces deux versions sont complètement implémentées en C#. Pour cela, nous nous basons sur le SDK Xamarin et sur un market de composants Xamarin. Dans chacune des versions, nous avons implémenté spécifiquement l'interface utilisateur de l'application. Pour cela, nous nous sommes basés sur les représentations en C# des SDKs Android et iOS. Ainsi, nous avons pu implémenter cette partie comme si nous l'avions fait nativement. Seul le langage a changé. Ensuite, les services indépendants de l'interface utilisateur ont été développés de façon commune en se basant sur le framework .Net fournit avec le SDK Xamarin. Cette partie du code est complètement partagée par l'application.

À la compilation, tout le code de l'application (interface utilisateur et code source commun) est transformé en code source natif. Ainsi l'application générée est considérée comme une application 100% native. Les règles de compilation de Xamarin sont relativement simples. En effet, vu que les SDK Android et iOS sont représentés en C# à travers Mono.Android.dll et Monotouch.dll sans aucune modification à part le changement de langage, il suffit, pour le compilateur de Xamarin, d'effectuer la transformation inverse, c'est-à-dire, de transformer le code C# en code natif sans aucune modification. Seul le langage change. Les instructions resteront toujours les mêmes. Bien sûr, cela concerne uniquement l'interface utilisateur qui est complètement liée aux implémentations natives des SDK Android et iOS. La partie com-

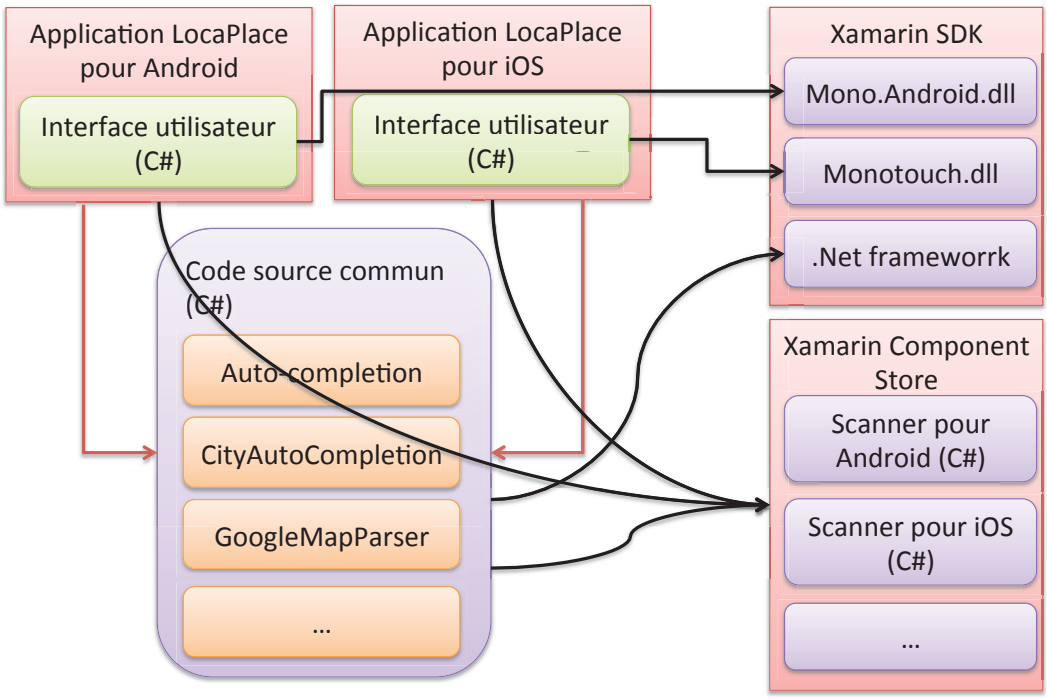


FIGURE 10.3 – Architecture de l'application LocaPlace implémentée avec Xamarin pour Android et iOS

mune du code source est transformée à partir d'autres règles de compilation. À l'exécution, aucun outil ne sera utilisé pour interpréter l'application.

Avec la présentation de l'architecture de LocaPlace à travers ces trois solutions, nous observons qu'elles sont très différentes dans leurs approches. PhoneGap embarque un site web qui est interprété à la volée par un interpréteur installé localement sur le smartphone. Titanium mobile lui embarque une application entièrement écrite en Javascript qui est aussi interprétée localement par un interpréteur à l'exécution. Quant à elle, Xamarin génère une application 100% native à partir d'un code source C#. Dans la suite, nous allons comparé ces trois solutions entre elles et avec COMMON notre framework de développement.

10.3 Résultats

Actuellement, nous avons implémenté le coeur de Locaplace, c'est-à-dire, les traitements effectués par nos composants présentés dans le chapitre 9, section 9.3, pour iOS et Android sous Titanium mobile et PhoneGap alors que seule la version Android a été développée avec Xamarin. Au niveau du développement de l'interface utilisateur, la première partie de l'application avec la recherche de lieux est fonctionnelle alors que la deuxième partie avec le scan de QR-Code n'a pas débuté. Nous n'avons pas non plus géré les erreurs possibles dans l'application. Par exemple, dans l'application générée avec Titanium mobile, si la localisation n'est pas activée sur le smartphone sur lequel l'application tourne, l'utilisateur sera bloqué sur une vue de chargement. Dans la prochaine version, une popup sera ouverte dans le but de demander à l'utilisateur d'activer sa localisation. Nous pourrions croire que cette partie est mineure dans le développement d'une application mobile. Cependant, c'est cette partie qui est souvent la plus longue à implémenter et qui est consommatrice en matière de lignes de code (environ 5000 lignes de code).

Avec l'implémentation du coeur de l'application, nous pouvons déjà présenter le ressenti dans l'utilisation de chacun des outils que ce soit par les développeurs et par les utilisateurs de l'application, section 10.3.1. Ensuite, nous comparons, avec chacune des solutions, les temps moyens d'exécution de chacun des processus déjà testés dans le chapitre précédent avec COMMON, section 10.3.2. Nous finirons par un début de comparaison sur la consommation de ressources de LocaPlace implémentée avec chacune des solutions.

10.3.1 Expérience utilisateur

Avant de comparer les performances de toutes les solutions testées, il est intéressant d'obtenir le ressenti des utilisateurs. Nous distinguons deux sortes d'utilisateurs : les développeurs d'applications qui utilisent les outils de développement proposés et les utilisateurs finaux qui utilisent l'application générée par les outils proposés.

Développeurs d'applications mobiles

Le développement de LocaPlace avec **PhoneGap** se fait comme un site web HTML classique. Pour le développer, nous avons utilisé un simple éditeur de texte pour la partie commune de l'application. Ensuite, en fonction de la cible sur laquelle nous voulons tester l'application, nous intégrons le code source dans un projet phoneGap sous eclipse ou Xcode. Pour chacun des IDEs, PhoneGap fournit un plugin qui permet de créer facilement un projet PhoneGap. Cependant, aucun outil n'est disponible pour vérifier que le code écrit est correct. Pour le vérifier, il faut lancer l'application sur un smartphone et vérifier que l'application se comporte comme ce qui était voulu. Ce processus est très lourd pour un développeur surtout lorsque l'on sait que le "time-to-market" des applications mobiles est très court. De plus, PhoneGap fournit l'accès aux couches basses des systèmes d'exploitation mobiles (géolocalisation, base de données, etc.). Le reste, c'est-à-dire l'interface utilisateur doit être développé à l'aide d'autres APIs complémentaires à phoneGap (JQuery mobile, etc.). Cependant, ces APIs ne sont pas développées par PhoneGap. Si un développeur veut les intégrer, il devra apprendre à les utiliser indépendamment de son application PhoneGap. Il devra se documenter en dehors du site web de PhoneGap, discuter avec d'autres développeurs de ces APIs en dehors de la communauté PhoneGap, etc. La complémentarité avec d'autres outils web est en même temps une force et une faiblesse de PhoneGap. C'est une force car ce système offre une multitude de fonctionnalités supplémentaires mais c'est aussi une faiblesse car c'est au développeur d'aller, pour chacun des outils, vérifier son fonctionnement. Cela peut prendre beaucoup de temps en fonction de la complexité de l'API à étudier. De plus, il se pourrait que certaines des APIs complémentaires à PhoneGap ne soient pas compatibles entre elles. Enfin, en implémentant PhoneGap, nous avons remarqué qu'il est parfois nécessaire d'écrire certaines parties du code dans la partie native de l'application générée. En effet, dans la présentation de PhoneGap, nous avons expliqué que le "site web" est encapsulé dans un projet natif pour Android et iOS. Ce sont ces deux projets que nous avons dû modifier en fonction de nos besoins (gestion de bases de données). Par conséquent, en plus de connaître les technologies web et les différents plugins compatibles avec PhoneGap, le développeur devra avoir des connaissances dans la programmation native.

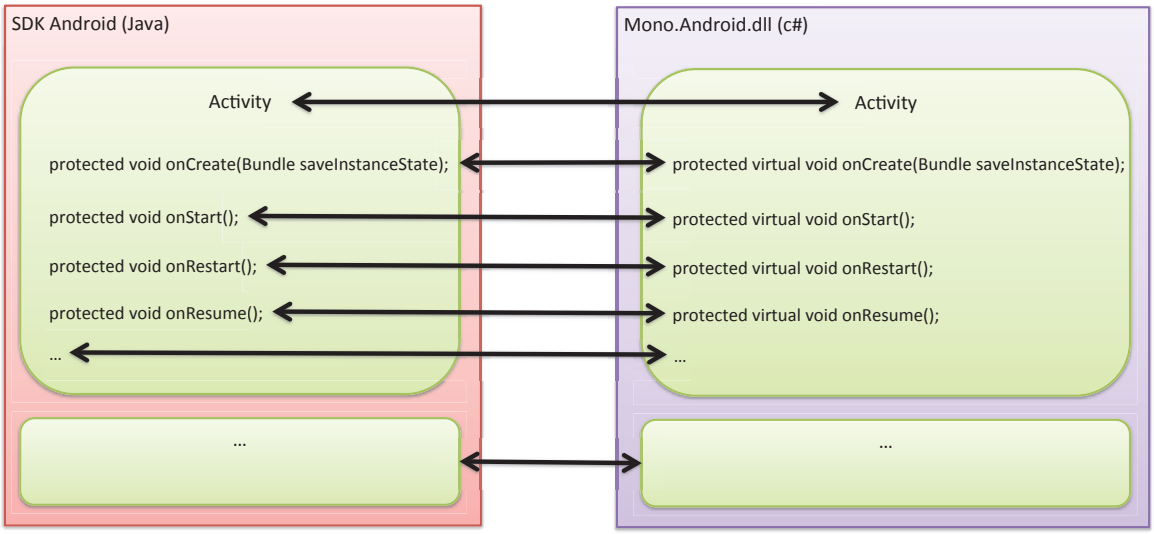
L'implémentation de LocaPlace avec **Titanium mobile** a été laborieuse. Ceci s'explique en deux temps. D'une part, l'IDE Titanium studio ne permet pas à un développeur d'être productif. D'autre part, les APIs provenant du framework Alloy ou du SDK Titanium ne se comportent pas de la même façon en fonction de la plate-forme cible. L'IDE Titanium studio est basé sur eclipse. Cependant, il n'intègre pas un outil de correction syntaxique de

code Javascript ou XML performant. Les erreurs sont souvent captées à la compilation ou à l'exécution quand l'application plante. Cela oblige les développeurs à tester fréquemment leurs applications. Cependant, le test d'une application Titanium mobile prend beaucoup de temps. L'outil de développement met, en effet, plus d'une minute à chaque compilation de l'application pour une seule plate-forme. Cette minute comprend le temps de packaging de l'application puis l'utilisation des outils natifs pour déplacer l'application générée sur un smartphone. Le temps d'une minute à la compilation est observé sur un Mac qui a un processeur Intel Core i7, 2,7 GHz et 16 Go de RAM. Ensuite, les APIs utilisées dans l'application ne se comportent pas toujours de la même façon. Par exemple, il est impossible de définir une seule fois l'interface graphique pour Android et iOS. Les éléments du framework Alloy se comportent d'une façon très différente sur Android et iOS. Ces différences n'impactent pas uniquement l'interface utilisateur. Elles posent des problèmes dans la gestion des évènements liés aux interactions utilisateurs. Un champ texte, et plus particulièrement le gain ou la perte de focus, ne se comporte pas du tout de la même façon sur Android et iOS. Ces différences se retrouvent aussi dans l'utilisation du SDK Titanium. Titanium mobile a donc réussi à unifier les appels aux fonctionnalités offertes par le mobile. Cependant, l'interprétation est différente en fonction des plates-formes visées. Cela implique donc de tester souvent l'application sur toutes les plates-formes cibles. Comme il est indiqué précédemment, les tests sont très longs à effectuer. Ces contraintes sont difficiles à gérer pour un développeur. Enfin, lorsque l'application est exécutée sur un smartphone iOS, Titanium mobile n'affiche pas les logs liés à l'application. Dans ce cas, le débogage devient impossible.

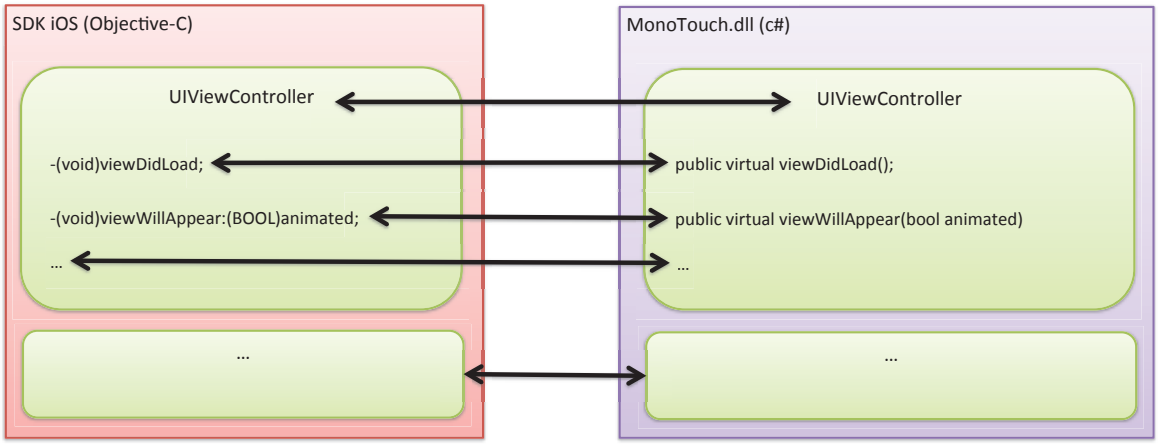
Xamarin est facile à utiliser pour un développeur mobile déjà confirmé. En effet, pour implémenter l'interface, il suffit de transposer le code qu'il aurait écrit en Java ou Objective-C en C#. Les méthodes à utiliser sont exactement les mêmes. Seul le langage change. Par exemple, sur Android, le cycle de vie d'une vue (Activity) est représenté par plusieurs méthodes natives dont ***protected void onCreate(Bundle savedInstanceState) ;***, ***protected void onStart() ;***, ***protected void onRestart() ;*** et ***protected void onResume() ;***, etc.. Ces méthodes seront exactement les mêmes en C# avec Mono.Android.dll, figure 10.4a. Pour implémenter le cycle de vie d'une vue, le développeur aura donc à sa disposition les mêmes méthodes en C# : ***protected virtual void OnCreate (Bundle savedInstanceState) ; protected virtual void OnStart () ;***, ***protected virtual void OnRestart () ;***, ***protected virtual void OnResume () ;***, etc.. Le seul effort à effectuer est donc d'apprendre le développement en langage C#. Cependant, pour les développeurs qui débutent dans le développement mobile, cette approche est plus difficile à appréhender. Toujours en prenant l'exemple du cycle de vie d'une vue, figure 10.4b, sur iOS cette fois, le développeur aura à sa disposition les méthodes ***public virtual void ViewDidLoad () ;***, ***public virtual void ViewWillAppear (bool animated) ;*** etc. provenant de Monotouch.dll et qui représentent le cycle de vie d'une vue développée nativement (UIViewController) : ***-(void)viewDidLoad ;***, ***-(void)viewWillAppear :(BOOL)animated ;***, etc.. Dans ce cas, le développeur débutant pourrait se demander pourquoi en fonction du système d'exploitation l'implémentation du cycle de vie change. Il ne pourra donc pas s'appuyer uniquement sur le développement Xamarin pour développer ses applications. Il devra se former sur Android et iOS avant d'utiliser Xamarin. À part ceci, l'outil de développement Xamarin sutdio est performant. Il fournit toutes les fonctionnalités que l'on peut attendre d'un IDE classique.

Utilisateurs finaux

Après avoir donné le ressenti constaté par les développeurs au moment de l'implémentation de chacune des versions de LocaPlace, il est intéressant de récupérer celui des utilisateurs finaux. Tout d'abord, **Phonegap** ne fournit pas la fluidité que l'on retrouve avec les appli-



(a) Liaison entre le SDK Android et Xamarin (Mono.Android.dll)



(b) Liaison entre le SDK iOS et Xamarin (MonoTouch.dll)

FIGURE 10.4 – Liaisons entre les SDKs natifs et les SDKs de Xamarin

cations natives. Même si l'interface a un aspect natif grâce aux librairies intégrées (JQuery mobile), le comportement global de l'application s'apparente plus à un site web que l'on exécuterait dans le navigateur web du smartphone. Par exemple, le chargement des pages web se ressent lors de l'utilisation. Il faut parfois une seconde avant qu'une page soit chargée alors que les pages web sont stockées localement sur le smartphone. De la même façon, les animations et transitions entre les différents éléments de l'interface se font parfois de façon laborieuse (freeze de l'écran). L'outil qui interprète l'application est donc trop peu performant pour que son impact soit transparent pour les utilisateurs. Malgré ce problème de fluidité, globalement, l'application fonctionne correctement sur Android et iOS.

La version implémentée avec **Titanium mobile** pour iOS et Android a l'aspect d'une application complètement native (interfaces et comportements). Cependant, elle n'offre pas l'aspect dynamique attendu avec les applications natives. Par exemple, dans LocaPlace sur la page de sélection des types et lieux des points d'intérêts à retrouver, figure 10.5, nous cachons des éléments de la page en fonction des interactions utilisateurs captées. L'objectif est de se focaliser sur la recherche que l'utilisateur effectue et de ne pas le polluer avec des informations inutiles. Ainsi, sur le premier état de la page, figure 10.5, les deux menus de sélection (catégorie et ville) sont disponibles. Lorsque l'utilisateur clique sur le champ "Autour de moi", nous

cachons alors tout ce qui est lié à la selection de la catégorie. Ensuite, sur le troisième état de la page, l'utilisateur entre le nom de sa ville. Une liste s'ouvre alors, avec toutes les villes correspondantes à ce qu'il a tapé. Lorsqu'il clique sur une des villes de la liste, le champ ville est automatiquement rempli et la vue retourne à son état initial. Bien sûr, le fait de cacher des éléments se fait à l'aide d'animations. Avec Titanium mobile, cette cinématique ne peut pas être mise en oeuvre sans que la page freeze. Un autre exemple est l'utilisation du swipe. Avec Titanium mobile rien n'existe pour cela. Certaines interactions utilisateurs n'ont donc pas pu être implémentées comme il l'était prévu au départ. De plus, l'application plante relativement souvent sur certains smartphones ou tablettes. En regardant les logs liés aux différents crashs qui peuvent survenir, nous observons que ce n'est pas notre application qui crash mais l'outil qui l'interprète à l'exécution. Ces différents problèmes et manques ont pour effet d'appauvrir l'expérience utilisateur et par la suite d'obtenir rapidement un rejet des utilisateurs.

Comme Titanium mobile, la version avec **Xamarin** a l'aspect d'une application native. Cela s'explique principalement par le fait que l'interface utilisateur est développée comme elle l'aurait été nativement. En matière d'ergonomie, l'application peut donc offrir toutes les fonctionnalités qui sont présentes à travers le développement mobile que l'on soit sur Android ou sur iOS. Par conséquent, sur Android, nous avons implémenté une application qui suit entièrement ce qui était prévu au départ sur le storyboard présenté dans le chapitre 9, section 9.2. La version iOS n'est pas encore implémentée. Après avoir utilisé l'application en condition réelle, nous en avons conclu que la version implémentée avec Xamarin pour Android fournit une expérience utilisateur similaire à celle d'une application native.

Avec nos premières constatations, PhoneGap et Titanium mobile ne répondent pas à nos besoins en matière d'IHM et d'environnements de développement. Pour valider cela, dans la suite de nos travaux, nous ferons tester chacune des applications par plusieurs testeurs. Nous leur demanderons ensuite de remplir un questionnaire à choix multiples sur l'expérience utilisateur ressentie à travers chacune des versions de l'application (lenteur, ralentissement, navigation, fluidité ...). Nous pourrons ainsi vérifier que ce que nous constatons en tant qu'utilisateur et développeur confirmé le soit aussi par des utilisateurs lambdas.

10.3.2 Performances

Dans le but de formaliser nos ressentis, nous avons récupéré les performances des trois versions de l'application (PhoneGap, Titanium mobile et Xamarin) sur tous les téléphones présentés dans le chapitre 9, section 9.4.3. Ainsi, nous pouvons comparer les huit processus présentés dans le chapitre 9, section 9.4.3. Cependant pour des raisons de lisibilité, dans ce chapitre, nous limitons notre étude des performances sur un seul smartphone Android et iOS et uniquement à six des huit processus présentés dans le chapitre précédent. Il faut uniquement savoir que les résultats ont les mêmes tendances sur tous les smartphones utilisés.

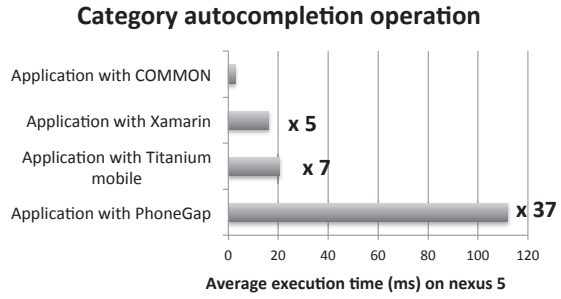
Sur Android

Sur la figure 10.6, nous avons représenté les temps moyens d'exécution de six processus relevés dans les différentes versions de LocaPlace : PhoneGap, Titanium mobile, Xamarin et COMMON. Dans chacune des configurations, la version implémentée avec COMMON est plus performante. En particulier, l'accès aux fichiers et le parsing des réponses aux web services sont nettement meilleurs. PhoneGap fournit, en effet, des temps 37 fois plus lents pour l'accès aux fichiers et 1,9 fois à 2,8 fois plus lents pour le parsing de réponses XML. La formation d'un itinéraire est aussi 2,9 fois plus lente qu'avec COMMON. Sur Titanium mobile, l'accès

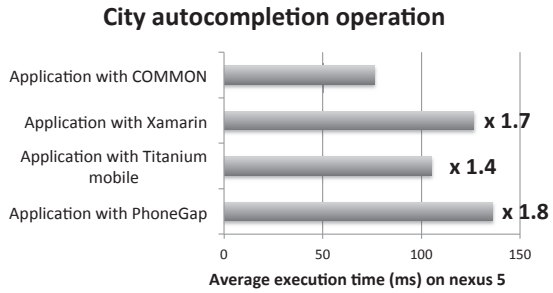
FIGURE 10.5 – Cinématique de la sélection d'une ville avant de lancer la recherche de lieux dans l'application LocaPlace

au fichier est problématique, 7 fois plus lent, ainsi que le parsing de réponses XML qui est 2,8 fois à 9,3 fois plus lents qu'avec COMMON. Xamarin a les mêmes problèmes. L'accès au fichier prend 5 fois plus de temps qu'avec COMMON et le parsing des réponses aux web services est 2 à 2,2 fois plus lent.

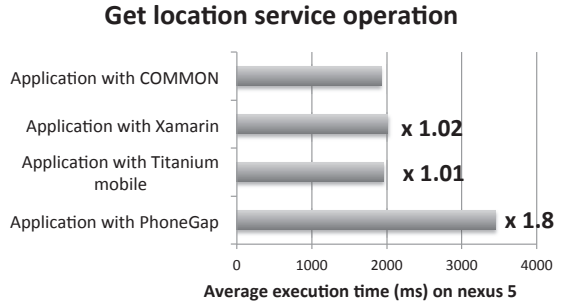
Au vu des résultats, la solution la plus stable, après COMMON, est Xamarin, alors que la solution la moins stable est PhoneGap. Cela s'explique naturellement par le fait que le code source fourni par Xamarin est complètement natif, alors qu'avec les autres solutions, il est interprété en temps réel. Néanmoins, l'écart entre Xamarin et les autres solutions n'est pas significatif. Nous supposons que le code généré avec Xamarain n'est pas encore optimal.



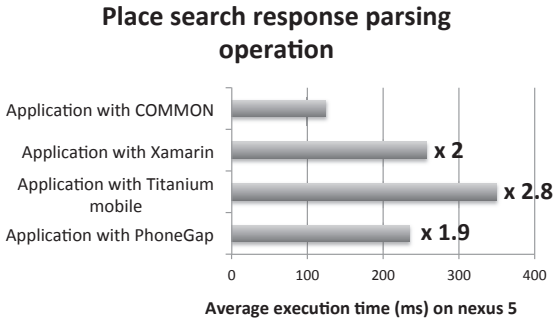
(a) Temps d'exécution moyen relevé lors de l'autocomplétion à partir d'un fichier texte



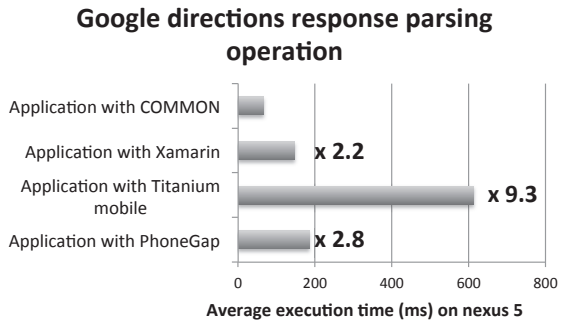
(b) Temps d'exécution moyen relevé lors de l'autocomplétion à partir d'une base de données



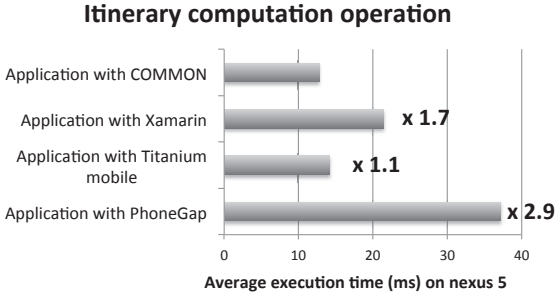
(c) Temps d'exécution moyen relevé lors de la récupération de la localisation de l'utilisateur



(d) Temps d'exécution moyen relevé lors du parsing de la réponse reçu du web service Place Search API



(e) Temps d'exécution moyen relevé lors du parsing de la réponse reçu du web service Google directions



(f) Temps d'exécution moyen relevé lors du parsing de la formation d'un itinéraire

FIGURE 10.6 – Comparaison des temps d'exécution moyens relevés sur un nexus 5 à travers les différentes versions de l'application LocaPlace

Sur iOS

Sur iOS, seules les performances des versions implémentées avec PhoneGap et Titanium mobile ont été relevées, figure 10.7. En effet, avec Xamarin nous n'avons pas implémenté l'interface graphique de la version iOS. Par conséquent, il est impossible pour nous de lancer les tests.

Comme sur Android, les résultats relevés montrent que l'application LocaPlace implémentée avec COMMON est plus performante que celles implémentées avec PhoneGap et Titanium mobile. Cependant, ici, les facteurs de temps supplémentaires sont beaucoup plus importants que sur Android. Seul l'accès à une base de données et au service de localisation offre des performances négligeables, inférieur à 2 fois, figures 10.7b et 10.7c. Sur Titanium mobile, le parsing des réponses retournées par les web services est beaucoup trop lent, 20 à 22 fois plus lent, figures 10.7d et 10.7e. Ce type d'opérations est très courant dans le mobile. Il est donc indispensable d'être au moins aussi performant que le développement natif sur ce point. Avec Titanium mobile, il faut attendre plus d'une demi-seconde pour parser les réponses du web service Google directions. L'accès aux systèmes de fichiers et la formation d'un itinéraire sont aussi très lents, 10 et 12 fois plus lent, figures 10.7a et 10.7f. Sur PhoneGap, les résultats sont plus nuancés. Le parsing des réponses à des web services est 2 à 3 fois plus lent. Seul l'accès au fichier et la formation de l'itinéraire sont très lents, 46 fois plus lent, figures 10.7a et 10.7.

Ces mauvais résultats s'expliquent principalement par le fait qu'Apple à travers iOS privilégie le code 100% natif alors que PhoneGap et Titanium mobile fournissent un moteur d'exécution qui interprète en temps réel le code source. Même si ce n'est pas une machine virtuelle, on s'en rapproche fortement. PhoneGap et Titanium mobile offrent de meilleures performances sur Android par rapport à iOS. Normalement, Xamarin devrait avoir des meilleures performances par rapport à ces deux solutions.

Dans cette comparaison, nous avons exclu deux processus : l'envoi de requêtes HTTP et le parsing des réponses du web service Place Details. Dans chacune des versions implémentées, l'envoi de requêtes HTTP fournit les mêmes performances que notre framework COMMON. Dans le cas du parsing des réponses du web service Place Details, ce processus est similaire au processus de parsing des réponses du web services Place Search. En effet, pour les deux processus, nous obtenons les mêmes facteurs de temps supplémentaire.

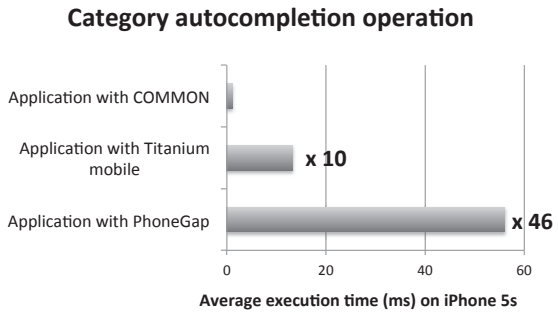
10.3.3 Utilisation des ressources

Aujourd'hui, pour chacune des solutions testées, nous avons récupéré la RAM utilisée avec les APIs fournies par chaque technologie. Cependant, nous n'avons pas réussi à interpréter et comparer correctement les valeurs retournées.

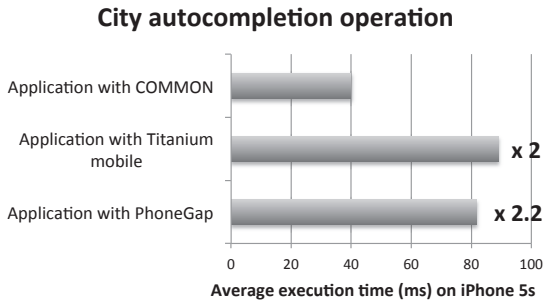
Sur Android, nous avons pu contourner ce problème en passant par une application tierce : **Smart Booster** [2]. Cette application permet de récupérer la RAM utilisée par chaque application en cours d'exécution sur un smartphone Android, figure 10.8. Nous n'avons donc pas pu obtenir la RAM moyenne consommée après chaque processus comme dans le précédent chapitre. Néanmoins, nous avons quand même observé la RAM utilisée par l'application à la fin d'une utilisation complète (autocomplétion, localisation, recherche de lieu, recherche de plus d'informations sur le lieu, recherche d'un itinéraire), figure 10.8.

Nous avons effectué le même parcours avec chaque version Android de l'application et

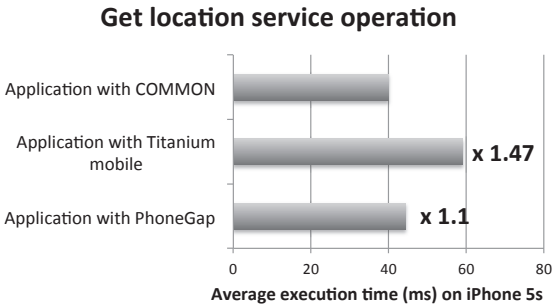
2. Site de téléchargement de l'application Smart Booster : https ://-play.google.com/store/apps/details ?id=com.rootuninstaller.rambooster&hl=fr_FR



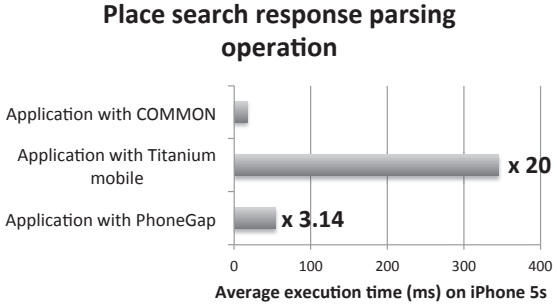
(a) Temps d'exécution moyen relevé lors de l'autocomplétion à partir d'un fichier texte



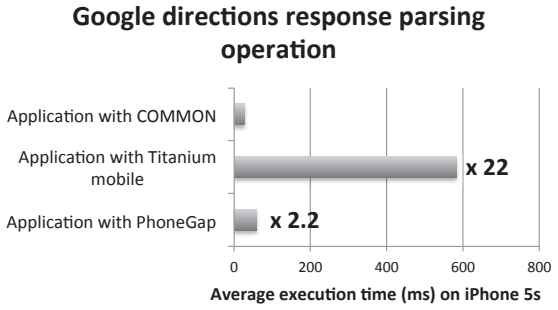
(b) Temps d'exécution moyen relevé lors de l'autocomplétion à partir d'une base de données



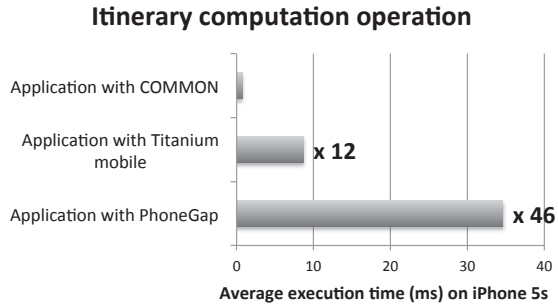
(c) Temps d'exécution moyen relevé lors de la récupération de la localisation de l'utilisateur



(d) Temps d'exécution moyen relevé lors du parsing de la réponse reçue du web service Place Search API



(e) Temps d'exécution moyen relevé lors du parsing de la réponse reçue du web service Google directions



(f) Temps d'exécution moyen relevé lors du parsing de la formation d'un itinéraire

FIGURE 10.7 – Comparaison des temps d'exécution moyens relevés sur un iPhone 5s à travers les différentes versions de l'application LocaPlace

nous avons comparé les résultats relevés dans le tableau 10.1. Nous observons que la version Android implémentée avec COMMON est moins consommatrice de RAM par rapport aux versions PhoneGap, Titanium mobile et Xamarin. Contrairement à ce qu'on aurait pu croire, l'application avec Xamarin est très consommatrice en RAM malgré le fait que le code généré soit complètement natif. Dans le cas de Xamarin, il n'y a aucun moteur d'exécution supplémentaire.

Sur iOS, il n'était pas possible d'effectuer la même comparaison. Dans la suite de nos travaux, nous voudrions analyser plus finement la RAM utilisée. Nous implémenterons nous-mêmes notre application tierce que nous appellerons à la demande depuis LocaPlace.

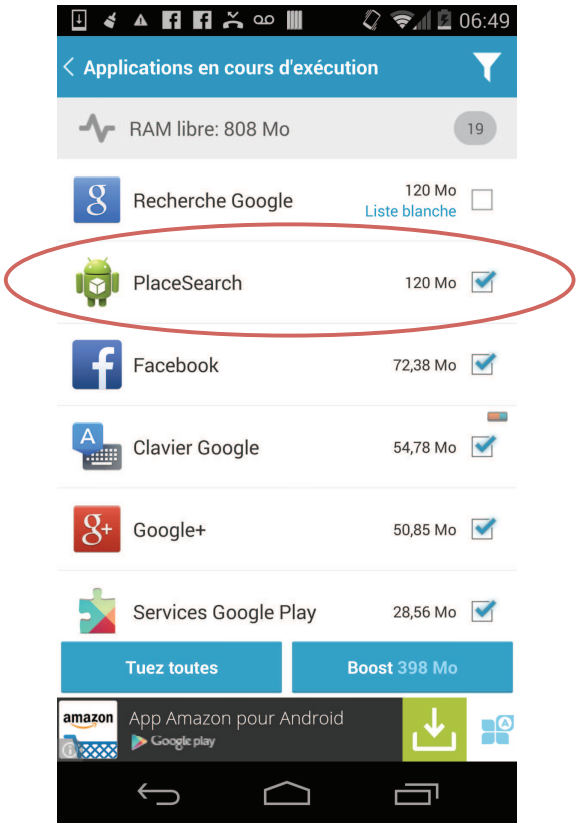


FIGURE 10.8 – Récupération de la RAM utilisée de l'application LocaPlace implémentée avec COMMON à travers l'application Smart Booster

TABLE 10.1 – RAM utilisée par chaque version de l'application LocaPlace après une utilisation complète sur un nexus 5

Solutions testées	RAM utilisée sur un nexus 5
Application implémentée avec PhoneGap	166 Mo
Application implémentée avec Titanium mobile	130 Mo
Application implémentée avec Xamarin	159 Mo
Application implémentée avec COMMON	120 Mo

Pour finir cette étude, nous avons pour chaque version de l'application donné son poids après avoir été installée sur un smartphone Android ainsi que sur un smartphone iOS, tableau

10.2.

TABLE 10.2 – Poids de chaque version de l'application LocaPlace après une installation sur Android et iOS

Solutions testées	Poids de l'application sur un nexus 5	Poids de l'application sur iOS
Application implémentée avec PhoneGap	14,63 Mo	12,4 Mo
Application implémentée avec Titanium mobile	27,75 Mo	6,9 Mo
Application implémentée avec Xamarin	17,59 Mo	Non implémentée
Application implémentée avec COMMON	7,17 Mo	4,9 Mo

Malgré le fait que l'application ne soit pas encore développée complètement avec chacune des solutions, chaque version est plus lourde en matière d'espace disque à l'installation par rapport à COMMON. Avec PhoneGap et Titanium mobile, cela s'explique par le fait qu'en plus du code écrit par les développeurs mobiles, ces solutions intègrent un moteur d'exécution avec l'application. Ce surplus n'est pas négligeable. En effet, si tous les développeurs implémentent leurs applications avec PhoneGap ou Titanium mobile, chacune d'elles intégrera ce moteur d'exécution. Sur chaque smartphone, il y aurait donc potentiellement plusieurs fois le même moteur d'exécution installé ce qui peut, à la longue, prendre beaucoup de place inutile. Pour être plus optimal, il aurait fallu installer le moteur d'exécution une seule fois sur chaque smartphone. Ce procédé est interdit sur iOS. Avec Xamarin, l'application est aussi plus lourde. Cela est plus surprenant. Effectivement, le code généré sur Android ne doit pas encore être optimal ou les librairies communes que Xamarin intègrent sont très lourdes par rapport à leur utilisation finale.

10.3.4 Discussions

Dans le tableau 10.3, nous avons récapitulé cette étude à travers nos besoins présentés dans le chapitre 2, section 2.5 et dans le chapitre 5, section 5.1. Dans ce chapitre, nous n'avions pas parler du critère "Évolutions des applications existantes". Il correspond à l'action de faire évoluer un projet déjà déployé sur les magasins d'applications officiels de chaque plateforme cible. Aujourd'hui, chez Keyneosoft et dans les entreprises de développement mobile en général, nous avons déjà implémenté nativement un grand nombre d'applications que nous maintenons et faisons évoluer depuis quelques années. Il est, pour nous, primordial en choisissant ce genre de technologies de développement qu'elles nous permettent de faire évoluer nos applications existantes. Aujourd'hui, seul COMMON permet de le faire.

Aujourd'hui, deux approches sont constamment opposées. Le web versus le développement natif [MT13, CSS12, CL11]. Ce constat n'est pas uniquement fait dans le monde académique mais aussi dans le monde industriel. Nous recevons de plus en plus de demande pour un site web ou une application web pour mobile au lieu d'implémenter plusieurs versions de la même application. Dans la comparaison que nous avons effectué, nous pouvons considérer que Phonegap et Titanium mobile font partie du côté web alors que Xamarin et COMMON font partie du côté natif. D'après nos résultats et notre expérience, le développement natif

offre clairement de meilleures performances, est plus fiable et offre plus de possibilités que le développement basé sur le web.

10.4 Conclusion

Dans nos futurs travaux, nous complèterons les trois versions de LocaPlace implémentées avec PhoneGap, Titanium mobile et Xamarin. L'objectif final sera de fournir la même évaluation que celle présentée dans le chapitre précédent. Ainsi, nous mettrons en avant le nombre de lignes de code gagnées avec chaque solution. Finalement, nous terminerons la comparaison sur la consommation des ressources en ajoutant la consommation de RAM utilisée pour chacun des processus testé.

Avec cette première comparaison, nous avons montré que COMMON fournit de meilleures performances que les solutions existantes PhoneGap, Titanium mobile ou Xamarin que ce soit en matière de temps d'exécution ou de consommation de ressources. De plus, l'application implémentée avec COMMON fournit la même expérience utilisateur qu'une application 100% native. Sur ce point Xamarin est équivalent alors que PhoneGap et Titanium mobile ne permettent pas d'avoir une expérience utilisateur optimale. COMMON est aussi plus flexible que ces trois technologies. En effet, avec COMMON, nous pouvons faire évoluer les applications déjà développées nativement sans pour autant perdre en performance. Dans le cadre d'une entreprise mature dans le développement mobile, COMMON pourrait être l'outil idéal.

Dans la prochaine partie, nous concluons cette thèse et donnons les perspectives découlant de ce projet.

Table 10.3 – Comparaison de chaque solution par rapport à nos besoins

Solutions	Expérience utilisateur	Outils de développement	Performances	Consommation de ressources	Évolutions des applications existantes	Un seul code source
PhoneGap	Similaire à un site web	Outils très faible (pas de débogeur)	Très inférieure au développement natif (réf COMMON). Se ressent sur l'expérience utilisateur.	Consommation excessive de RAM + Moteur d'exécution lourd	Impossible	1 seul code source avec une différenciation de l'IHM si besoin
Titanium mobile	Ressemble à du natif (éléments graphiques, animations) mais n'est pas aussi riche que le natif (aspect dynamique)	IDE basé sur eclipse sans correcteur syntaxique performant, sans débogueur	Très inférieure au développement natif (réf COMMON)	Moteur de d'exécution très lourd sur Android. Consommation de RAM correcte	non	1 seul code source différencier en fonction des plates-formes lorsque les APIs n'ont pas le même comportement
Xamarin	Identique au natif (développement comme en natif)	Aussi performant qu'un IDE classique	Une à cinq fois supérieurs (souvent négligeable)	Application plus lourde. Consommation excessive de la RAM	non	2 codes sources séparés avec une base commune
COMMON	Identique au natif (développement natif)	IDE officiels	Identique au natif	Identique au natif	oui	2 codes sources séparés avec une base commune

Quatrième partie

Conclusions et perspectives

Chapitre 11

Conclusion et Perspectives

Dans ces travaux de thèse, nous avons proposé une architecture hybride basée sur la programmation par composants pour le développement d'applications mobiles multiplateformes. Pour ce faire, nous avons introduit la notion de composants multiplateformes pouvant être intégrés dans n'importe quelle application que ce soit Android, iOS, Windows Phone 8, etc. Nous avons ensuite montré, à travers le développement d'une application mobile, que notre solution permettait de gagner 30% de code source tout en gardant les mêmes fonctionnalités que le développement natif (même expérience utilisateur, même performance, etc.). Nous avons aussi comparé notre solution à des produits disponibles sur le marché. Nous en avons conclu que c'était la seule qui répondait à tous nos besoins sans aucune limitation pour les développeurs. Dans ce chapitre, nous concluons ces travaux et donnons quelques perspectives possibles suite à ce travail.

11.1 Conclusion

Avec l'essor du mobile, le développement d'applications pour la mobilité est devenu un marché à part entière de l'informatique. De plus en plus d'entreprises et développeurs indépendants se spécialisent dans l'édition de logiciels pour smartphones. Cependant, ils se heurtent à un frein majeur dans le déploiement de leurs produits. En effet, l'hétérogénéité des systèmes d'exploitation est devenue un vrai problème.

Au lancement des smartphones et des magasins d'applications, les applications implémentées étaient souvent anecdotiques. L'objectif était de se montrer sur les magasins d'applications avec ces applications, faisant le buzz. Il fallait généralement quelques semaines pour les implémenter et ensuite les publier. Dans ce cas, il était encore possible d'implémenter la même application pour plusieurs plates-formes mobiles sans obligatoirement que le coût soit très élevé. Depuis quelques temps, avec l'augmentation des capacités des smartphones, la tendance est plutôt au développement d'applications mobiles complexes. Il faut maintenant quelques mois pour qu'une application soit conçue, implémentée et publiée. Vu ce changement, il est devenu nécessaire de baisser le coût de développement d'une application multiplateforme.

Dans ces travaux de thèse, nous nous sommes intéressés à cette problématique. Pour y répondre, nous avons proposé d'utiliser et d'adapter la programmation par composants pour le développement d'applications mobiles multiplateformes. En effet, la programmation par composants permet de baisser le coût de développement d'un logiciel en favorisant la réutilisation des composants. De plus, ce type de programmation permet d'augmenter la qualité des logiciels produits et facilite la maintenance.

Quelques modèles à composants ont été portés sur mobile et plus particulièrement sur Android. Cependant, aucun n'a été spécifié pour le développement multiplateforme. Effectivement, dans ces modèles, les composants sont implémentés pour un environnement spécifique. Ensuite, l'assemblage de composants se fait à travers un langage de description souvent très proche de la plate-forme de développement. Ceci, implique que pour chaque environnement, il faut développer le composant et le décrire en fonction de la plate-forme de développement. Un composant JAVA pour Android n'est pas décrit de la même façon qu'un composant Objective-C pour iOS. Enfin, l'assemblage sur chaque plate-forme est différent car les descriptions des composants sont différentes.

Tout d'abord, nous avons combiné le développement natif avec la programmation par composants. Effectivement, dans notre solution, l'assemblage de composants se fait à travers le code source d'une application mobile développée nativement pour n'importe quelle plate-forme mobile. Ce choix implique que les développeurs d'applications mobiles doivent avoir plusieurs versions de leurs applications en fonction du nombre de plates-formes cibles. Par exemple, pour une application Android et iOS, le développeur aura obligatoirement deux versions de son application, une implémentée en JAVA pour Android et l'autre implémentée en Objective-C pour iOS. Ainsi, en laissant la possibilité d'implémenter une partie de l'application nativement, les développeurs peuvent réaliser l'ergonomie de leurs applications de façon native. L'application finale aura alors un aspect et un comportement propre à chaque plate-forme cible. Aujourd'hui, dans les produits sur le marché, le principal défaut est qu'ils diminuent l'expérience utilisateur, notamment parce que l'ergonomie n'est pas native. De plus, il se peut que le panel de composants de notre solution ne corresponde pas à tous les besoins, c'est-à-dire, que pour une fonctionnalité donnée, aucun composant n'y réponde. Dans ce cas, soit le développeur de l'application implémente son propre composant, soit il développe la fonctionnalité manquante nativement. Vu que le développement d'un composant peut être coûteux et mettre un certain temps à être rentabilisé, parfois en fonction des contraintes d'un projet, la deuxième solution sera plus efficace.

C'est à partir de ces différents codes sources que les développeurs invoquent des composants (création d'une instance suivie d'un appel à une fonctionnalité). Pour ce faire, nous avons mis à disposition un langage commun à toutes les plates-formes mobiles. Ce langage est commun car il a la particularité de s'intégrer dans n'importe quel environnement de développement. Il est basé sur les annotations. Classiquement, les annotations sont utilisées pour ajouter des métadonnées à un élément d'un code source. Nous les avons détourné pour leur permettre d'appeler les méthodes d'un programme. En plus d'être commun à chaque plate-forme cible, notre langage permet de mutualiser les différentes invocations d'un composant. Cela signifie que, lorsque nous intégrons un composant sur une plate-forme, le même code source peut être réutiliser dans n'importe quel environnement de développement, que ce soit sur Android, iOS, etc. Pour ce faire, il nous fallait un seul point d'entrée pour l'accès à nos composants. Comme nous l'avons signalé, aujourd'hui, dans les modèles à composants existants un composant JAVA pour Android n'est pas décrit de la même façon qu'un composant Objective-C pour iOS. Par conséquent, nous avons deux points d'entrée, un pour Android, un pour iOS.

Pour pallier à ce manque, nous avons introduit la notion de composants multiplateformes.

Ce genre de composant contient plusieurs implémentations, une par plate-forme cible. Nous avons donc, par exemple, une implémentation Java pour Android et une autre en Objective-C pour iOS du même composant. Habituellement, un composant n'a qu'une seule implémentation. Le choix de différentes implémentations pour les composants nous est principalement dicté par l'architecture des plates-formes cibles. Nous ne pouvons effectivement pas exécuter un composant JAVA sur iOS. Ce procédé est aujourd'hui interdit et difficilement exploitable. Par conséquent, il nous faut une implémentation native de chaque composant pour chaque plate-forme cible. Nous avons ensuite proposé une nouvelle description de ce genre de composants à travers deux interfaces. La première interface est dite commune. Elle est indépendante d'une plate-forme cible. Pour ce faire, elle est focalisée uniquement sur l'aspect fonctionnel du composant. Elle met en avant les fonctionnalités et les paramètres fonctionnels associés et ne précise aucun détail technique sur son implémentation. La deuxième interface, est quant à elle, complètement dépendante de toutes les implémentations du composant. En effet, si le composant est implémenté pour Android et iOS, cette interface contiendra la représentation de toutes les méthodes exposées du composant pour chacune des deux plates-formes.

Pour valider notre approche, nous avons implémenté les logiciels nécessaires à sa mise en oeuvre et son exploitation. Nous avons réalisé un générateurs d'interfaces pour nos composants ainsi qu'un interpréteur pour faciliter leurs intégrations. Ensuite, nous avons implémenté un compilateur permettant de transformer le code source écrit avec notre langage en code source pour Android ou iOS. Nous avons ensuite utilisé ces logiciels pour développer une application mobile nommée LocaPlace pour Android et iOS. Cette application intègre cinq composants multiplateformes. Ces composants exploitent tous les aspects que nous avons définis dans notre proposition. Chaque composant a été intégré avec notre langage commun. Tout le code source écrit avec notre langage a été transformé en langage natif par notre compilateur.

Nous avons comparé l'application LocaPlace, implémentée avec COMMON, avec la même application développée nativement sur Android puis sur iOS. Nous avons montré que sur chacune des plates-formes, les deux applications fournissent les mêmes performances et consomment les mêmes ressources. De plus, notre solution a permis de développer 32% de code en moins. Ce gain est considérable pour une entreprise. En plus de fournir des performances égales à celles relevées dans un cadre de développement purement natif, notre solution répond à tous nos besoins. En effet, avec elle, il est possible de fournir la même expérience utilisateur qu'une application native. De plus, nous ne fixons aucune limite au développement. Si une fonctionnalité est complètement spécifique à une plate-forme cible, il sera quand même possible de l'intégrer. Soit le développeur implémentera un composant, soit le développeur intégrera directement le code source nécessaire. Dans le premier cas, le composant sera intégrable sur une seule plate-forme mais sera réutilisable dans plusieurs applications. Enfin, notre solution s'intègre dans les applications mobiles existantes. Il est en effet possible d'intégrer des composants dans une application qui a déjà vécu. C'est un point fort de notre solution.

Pour finir, nous avons comparé notre solution à d'autres produits : Phonegap, Titanium mobile et Xamarin. Ces trois produits sont matures. Pour autant, ils ne répondent pas à tous nos besoins. De plus, nos premiers résultats ont montré que ces solutions fournissent des applications moins performantes que COMMON, que ce soit en temps d'exécution ou en consommation de ressources.

Pour récapituler, nos contributions ont permis :
— La combinaison du développement natif mobile avec la programmation par composants.
— L'unification de l'accès aux composants à travers un langage commun intégrable dans

n'importe quel environnement de développement.
— L'introduction de composants multiplateformes. Ces composants ont la particularité d'avoir plusieurs implémentations cachées à travers une interface indépendante de toutes les plates-formes cibles.

Ces travaux ont été validés à travers la publication de conférence nationale [PDL12] et internationale [PDL13] ainsi qu'une revue internationale [oPDL14].

11.2 Perspectives

Ces premiers travaux sur la prise en compte de l'hétérogénéité des terminaux mobiles pour diminuer le coût de développement des applications mobiles n'en sont qu'à leurs débuts. Ils ont ouvert plusieurs perspectives de recherches et industrielles à plus ou moins long terme.

Perspectives à court terme

Les résultats obtenus avec notre framework de développement n'ont pas été tous publiés. Nous n'avons pas encore publié la comparaison entre le développement natif et COMMON sur iOS. Ce point fera l'objet d'un article dans les mois à venir. De la même manière, nous aimerions publier la comparaison que nous avons effectuée entre COMMON et Phonegap, Titanium mobile et Xamarin. Cela implique de compléter l'implémentation de LocaPlace avec chacune des technologies. Nous pourrons ainsi comparer le nombre de lignes de code gagnées avec chaque solution.

Notre prototype COMMON est doté d'une dizaine de composants multiplateformes. Ces composants sont souvent de haut niveau. À court terme, nous allons réaliser et intégrer des composants de plus bas niveaux dans le but de couvrir plus de besoins. Cependant, le risque est de descendre trop bas et de ne pas pouvoir réunir les implémentations du composants entres elles. Dans ce cas, pour chaque plate-forme, il faudrait intégrer les composants d'une façon spécifique. Il est donc important de définir jusqu'où nous pouvons aller avec notre solution.

Perspectives à moyen terme

Pour la version initiale de notre solution, les annotations définies sont simples. Dans le futur, nous devons enrichir ce langage. Par exemple, une des pistes est de permettre la combinaison des annotations entre elles dans le but de faire des appels successifs à des composants. Une autre évolution pourrait être de lier les éléments de l'interface graphique directement aux composants. Le compilateur s'occuperait alors de générer le code source pour mettre à jour les éléments graphiques avec les résultats retournés par les composants. Une autre piste serait d'annoter nos propres annotations pour choisir le type de comportement que le composant doit suivre. Par exemple, l'invocation d'une méthode (***@method***) pourrait être annotée de ***@asynchronous*** pour signaler que le composant doit fonctionner en mode asynchrone. Ensuite, les composants réalisés pour le moment fournissent des services qui fonctionnent principalement en arrière plan. Plus tard, nous voulons réaliser des composants multiplateformes graphiques. En fonction du niveau d'abstraction du composant, l'intégration ne se fera pas obligatoirement dans le code source Java ou Objective-C mais aussi dans le code source graphique. Pour rappel sur Android ou Windows Phone, l'intégration se fait respectivement dans des fichiers XML ou XAML. Il serait intéressant d'unifier l'intégration de ce

genre de composants. L'objectif est bien sûr de couvrir tous les aspects du développement d'une application avec notre solution.

Dans la thèse, nous avons parlé de gestion contextuelle des paramètres non fonctionnels des fonctionnalités d'un composant au niveau de la compilation, chapitre 7, section 7.3.2. Au niveau de l'implémentation de notre compilateur, nous avons intégré cette gestion pour un seul type de paramètre (le contexte sur Android). Cependant, il se peut que les développeurs de composants aient besoin de déclarer d'autres types de paramètres comme non fonctionnels. Dans ce cas, nous voulons leur laisser la possibilité d'étendre eux-même les règles du compilateur pour gérer spécifiquement leurs paramètres non fonctionnels. Pour cela, pour chaque contexte de compilation, ils devraient décrire les valeurs à intégrer pour chaque paramètre non fonctionnel. Ce genre de traitement adresse de nouvelles problématiques telles que la définition d'un contexte de compilation, l'injection dynamique de règles dans un compilateur ou encore la comptabilité entre les règles définies pour un composant et pour un autre.

Les applications sensibles au contexte sont depuis quelques années mises en avant dans beaucoup de projets de recherche et en particulier dans les travaux portant sur la mobilité. Au LAMIH, nous avons nous-mêmes travaillé sur ce type d'applications [DLP+10, PDLP11]. L'objectif est de fournir des applications qui, en fonction du contexte, changent de comportement à l'exécution. Par exemple, une application de recherche d'itinéraire, si elle est utilisée dans une voiture sur l'autoroute, attendra les ordres du conducteur par reconnaissance vocale, alors que si la même application fonctionne lorsque l'utilisateur est à pied, dans la rue, elle n'attendra plus d'ordre vocal mais des ordres écrits textuellement. Dans les travaux de l'équipe Optimob du LAMIH, Popovici [PDLP11] a défini un cadre de développement d'applications mobiles sensibles au contexte. Ces travaux ont été déployés à travers un modèle à composants basés sur OSGi sur Android. Ainsi, en fonction du contexte, certains composants sont remplacés à l'exécution par d'autres composants sans pour autant redémarrer l'application. Cependant, aujourd'hui, cette plate-forme n'est disponible que pour Android. Dans l'avenir, il serait intéressant de reprendre les composants logiciels de CATS pour les intégrer à travers des composants multiplateformes. Ainsi, nous pourrions fournir des composants multiplateformes sensibles au contexte. Dans ce genre d'applications, nos annotations pourraient, en plus d'invoquer des méthodes, être utilisé pour définir la façon dont l'application doit se reconfigurer à l'exécution.

Perspectives à long terme

Notre solution est proche des systèmes d'exploitation. En effet, les composants sont implémentés pour chaque plate-forme avec les outils de développement natif. De même, les applications mobiles utilisant notre langage commun d'assemblage sont développées avec les environnements de développement natif et une partie du code qui est écrit nativement. Dans le chapitre 4, nous avons présenté des solutions de haut niveau, plus particulièrement l'IDM, permettant à partir d'une abstraction des plates-formes cibles le développement d'applications mobiles multiplateformes. Dans notre solution, même si les outils pour développer les composants et les intégrer sont de bas niveau, nous pouvons considérer nos composants comme des composants de haut niveau. Effectivement, ils fournissent des fonctionnalités de haut niveau dans le but de cacher la complexité nécessaire pour les implémenter. En partant de ce constat, il serait intéressant de lier les solutions de haut niveau avec nos composants. Ainsi, ces solutions pourront offrir des performances égales au développement natif en exploitant nos composants. De notre côté, nous pourrions implémenter une application avec un cadre de développement commun.

Une autre perspective de recherche qui n'a pas encore été (ou très peu) adressée dans

le domaine de la programmation par composants est la certification des composants. En effet, avec notre solution, n'importe qui est en mesure d'implémenter un composant et de le déployer. Cependant, avant l'intégration dans un logiciel il n'y aucun moyen de vérifier que le composant ne fournisse que les fonctionnalités souhaitées. Cette problématique s'applique plus généralement au domaine du mobile. Avant d'installer une application mobile, il est en effet difficile de savoir exactement ce qu'une application est en mesure d'effectuer (accéder à la liste des contacts, lire le contenu de la carte SD, etc.). Même si les actions possibles sont listées, il est difficile de savoir ce qu'une application va faire de ses droits. Il serait donc intéressant de pouvoir certifier un composant avant qu'il ne soit intégré dans une application.

Pour la thèse, nous avons implémenté un prototype de notre solution. Dans le futur, nous voulons l'industrialiser et le mettre en place à Keyneosoft. Pour ce faire, plusieurs solutions s'offrent à nous, soit le projet devient open-source à l'initiative de Keyneosoft et du LAMIH, dans ce cas, l'objectif serait de trouver une équipe de développeurs motivés pour faire vivre le projet. Soit l'outil est industrialisé chez Keyneosoft. Dans ce cas, nous pourrions, en plus de la mettre en place chez Keyneosoft, revendre cette solution. Pour le moment, aucun choix n'a été fait.

Bibliographie

[AGPK11] Nuno Amálio, Christian Glodt, Frederico Pinto, and Pierre Kelsen. Platform-variant applications from platform-independent models via templates. *Electr. Notes Theor. Comput. Sci.*, 279(3) :3–25, 2011.

[ANP11] Oren Antebi, Markus Neubrand, and Arno Puder. Cross-compiling android applications to windows phone 7. In *Mobile Computing, Applications, and Services - Third International Conference, MobiCASE 2011, Los Angeles, CA, USA, October 24-27, 2011. Revised Selected Papers*, pages 283–302, 2011.

[ASP06] Nuno Amálio, Susan Stepney, and Fiona Polack. A formal template language enabling metaproof. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006 : Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 252–267. Springer Berlin Heidelberg, 2006.

[AVHA+14] Jeremy Andrus, Alexander Van't Hof, Naser AlDuaij, Christoffer Dall, Nicolas Viennot, and Jason Nieh. Cider : Native execution of ios apps on android. *SIGPLAN Not.*, 49(4) :367–382, February 2014.

[AWR13] Paul Adenot, Chris Wilson, and Chris Rogers. Web audio api. w3c working draft. Technical report, W3C, October 2013.

[Bar09] Lee S. Barney. *Developing Hybrid Applications for the iPhone : Using HTML, CSS, and JavaScript to Build Dynamic Apps for the iPhone*. Addison-Wesley Professional, 1st edition, 2009.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, September 2006.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[BE08] P. Braun and R. Eckhaus. Experiences on model-driven software development for mobile applications. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 490–493, March 2008.

[BFH08] F. T. Balagtas-Fernandez and H. Hussmann. Model-driven development of mobile applications. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 509–512, Washington, DC, USA, 2008. IEEE Computer Society.

[BFL11] Gary Bennett, Mitch Fisher, and Brad Lees. Protocols and delegates. In *Objective-C for Absolute Beginners*, pages 261–265. Apress, 2011.

[BFTH10] Florence Balagtas-Fernandez, Max Tafelmayer, and Heinrich Hussmann. Mobia modeler : Easing the creation process of mobile applications for non-technical

users. In *Proceedings of the 15th International Conference on Intelligent User Interfaces*, IUI '10, pages 269–272, New York, NY, USA, 2010. ACM.

[BFV13] Abimael Barea, Xavier Ferre, and Lorenzo Villarroel. Android vs. ios interaction design study for a student multiplatform app. In Constantine Stephanidis, editor, *HCI International 2013 - Posters' Extended Abstracts*, volume 374 of *Communications in Computer and Information Science*, pages 8–12. Springer Berlin Heidelberg, 2013.

[Box97] Don Box. *Essential COM.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[BP08] Jean Bovet and Terence Parr. Antlrworks : an antlr grammar development environment. *Software : Practice and Experience*, 38(12) :1305–1332, 2008.

[Bro13] C. Brousseau. *Creating Mobile Apps with Appcelerator Titanium.* EBL-Schweitzer. Packt Publishing, 2013.

[Các12] Marcos Cáceres. Packaged web apps (widgets) - packaging and xml configuration (second edition). Technical report, W3C, November 2012.

[CCMSW01] Erik Christensen, Francisco Curbera, Greg Meredith, and IBM Research Sanjiva Weerawarana. Web services description language. Technical report, W3C, March 2001.

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS*, 15 :289–308, 2003.

[CDST13] Gaëlle Calvary, Thierry Delot, Florence SEDES, and Jean-Yves Tigli. *Computer Science and Ambient Intelligence.* Wiley, 2013.

[Cha13] Hassan Charaf. A methodology for model-driven multiplatform mobile applicaiton development. *International journal of computer engineering and technology (IJCET)*, 4 :61 – 70, January - February 2013.

[CJR12] Luis Corral, Andrea Janes, and Tadas Remencius. Potential advantages and disadvantages of multiplatform development framework. a vision on mobile environments. *Procedia Computer Science*, 10(0) :1202 – 1207, 2012. ANT 2012 and MobiWIS 2012.

[CL11] Andre Charland and Brian LeRoux. Mobile application development : Web vs. native. *Queue*, 9(4) :20 :20–20 :28, April 2011.

[CLY14] Chen-Fu Chien, Kuo-Yi Lin, and Annie Pei-I Yu. User-experience of tablet operating system : An experimental investigation of windows 8, ios 6, and android 4.2. *Computers and Industrial Engineering*, 73(0) :75 – 84, 2014.

[CR10] Daniele Cammareri and Claudia Raibulet. An enhanced approach to automatic generation of mobile widgets. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '10, pages 755–758, New York, NY, USA, 2010. ACM.

[CSS12] Luis Corral, Alberto Sillitti, and Giancarlo Succi. Mobile multiplatform development : An experiment for performance analysis. *Procedia Computer Science*, 10(0) :736 – 743, 2012. <ce :title>ANT 2012 and MobiWIS 2012</ce :title>.

[CZHNI+14] Belén Cruz Zapata, Antonio Hernández Niñirola, Ali Idri, JoséLuis Fernández-Alemán, and Ambrosio Toval. Mobile phrs compliance with android and ios usability guidelines. *Journal of Medical Systems*, 38(8), 2014.

[Dav12] Matthew David. Project : Building a native ios app with phonegap. In *HTML5 Mobile Websites*, pages 199 – 207. Focal Press, Boston, 2012.

[DDBN13] Isabelle Dalmasso, Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *IWCMC 2013, 9th IEEE International Wireless Communications and Mobile Computing Conference, July 1-5, 2013, Cagliari, Italy*, Cagliari, ITALIE, 07 2013.

[DLP+10] Mikael Desertot, Sylvain Lecomte, Dana Popovici, Marie Thilliez, and Thierry Delot. A context aware framework for services management in the transportation domain. In Khalil Drira, Ahmed Hadj Kacem, and Mohamed Jmaiel, editors, *NOTERE 2010, Annual International Conference on New Technologies of Distributed Systems, Touzeur, Tunisia, May 31 - June 2, 2010, Proceedings*, pages 157–164. IEEE, 2010.

[Doy14] Matt Doyle. *Master Mobile Web Apps with jQuery Mobile*. Elated Books, March 2014.

[dS03] Alberto Rodrigues da Silva. The xis approach and principles. In *29th EUROMICRO Conference 2003, New Waves in System Architecture, 3-5 September 2003, Belek-Antalya, Turkey*, pages 33–41. IEEE Computer Society, 2003.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 266–276, New York, NY, USA, 2002. ACM.

[EHL07] Clement Escoffier, Richard S. Hall, and Philippe Lalanda. ipojo : an extensible service-oriented component framework. *2013 IEEE International Conference on Services Computing*, 0 :474–481, 2007.

[EJB08] Programming weblogic enterprise javabeans, ejb 3.0 metadata annotations reference. Technical report, Oracle, August 2008.

[Fav06] J.M. Favre. *L'ingénierie dirigée par les modèles : au-delà du MDA*. IC2 : Série Informatique et systèmes d'information. Hermes Science Publications, 2006.

[Fra12] B. Frain. *Responsive Web Design with HTML5 and CSS3*. Community experience distilled. Packt Publishing, Limited, 2012.

[GBCP14] Olivier Le Goaer, Franck Barbier, Eric Cariou, and Samson Pierre. Android executable modeling : Beyond android programming. In *The 2014 International Workshop on Mobile Applications (MobiApps 2014).*, 2014.

[GG08] Vincenzo Gervasi and GiacomoA. Galilei. Software manipulation with annotations in java. In Egon Börger and Antonio Cisternino, editors, *Advances in Software Engineering*, volume 5316 of *Lecture Notes in Computer Science*, pages 161–184. Springer Berlin Heidelberg, 2008.

[GHG10] Tor-Morten Grønli, Jarle Hansen, and Gheorghita Ghinea. Android vs windows mobile vs java me : a comparative study of mobile development environments. In Fillia Makedon, editor, *PETRA*, ACM International Conference Proceeding Series. ACM, 2010.

[GHGY14] T.-M. Grønli, J. Hansen, G. Ghinea, and M. Younas. Mobile application platform heterogeneity : Android vs windows phone vs ios vs firefox os. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 635–641, May 2014.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GR11] Mark H. Goadrich and Michael P. Rogers. Smart smartphone development : ios versus android. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 607–612, New York, NY, USA, 2011. ACM.

[Här13] Marlo Häring. Platform architecture portfolio : Comparison of 3 platforms (android, ios, mobile ubuntu). In Stefan Wagner and Horst Lichter, editors, *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, 26. Februar - 1. März 2013 in Aachen*, volume 215 of *LNI*, pages 391–394. GI, 2013.

[HHM12a] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. Comparing cross-platform development approaches for mobile applications. In Karl-Heinz Krempels and José Cordeiro, editors, *WEBIST*, pages 299–311. SciTePress, 2012.

[HHM12b] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. Evaluating cross-platform development approaches for mobile applications. In José Cordeiro and Karl-Heinz Krempels, editors, *Web Information Systems and Technologies - 8th International Conference, WEBIST 2012, Porto, Portugal, April 18-21, 2012, Revised Selected Papers*, volume 140 of *Lecture Notes in Business Information Processing*, pages 120–138. Springer, 2012.

[HKM10] Sayed Hashimi, Satya Komatineni, and Dave MacLean. *Pro Android 2*. Apress, Berkely, CA, USA, 1st edition, 2010.

[HMCF+14] Dominique Hazaël-Massieux, Suresh Chitturi, Max Froumentin, Maria Angeles Oteo, and Niklas Widell. The messaging api. w3c working draft. Technical report, W3C, January 2014.

[HMK13] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. Cross-platform model-driven development of mobile applications with md2. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 526–533, New York, NY, USA, 2013. ACM.

[HMT+04] Assia Hachichi, Cyril Martin, Gaël Thomas, Simon Patarin, and Bertil Folliot. Reconfigurations dynamiques de services dans un intergiciel a composants CORBA CCM. *CoRR*, cs.NI/0411081, 2004.

[HV09] Zef Hemel and Eelco Visser. PIL : A platform independent language for retargetable DSLs. In Mark van den Brand and Jeff Gray, editors, *Software Language Engineering, Second International Conference, SLE 2009, Denver, USA, October, 2009. Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2009.

[HV11] Zef Hemel and Eelco Visser. Declaratively programming the mobile web with mobl. *SIGPLAN Not.*, 46(10) :695–712, October 2011.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Object-Oriented Programming*, 1(2), 1988.

[JFL10] Fan Jiang, Zhigang Feng, and Lei Luo. xface : A lightweight web application engine on multiple mobile platforms. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 2055–2060, Washington, DC, USA, 2010. IEEE Computer Society.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[KCO11] Dean Kramer, Tony Clark, and Samia Oussena. Platform independent, higher-order, statically checked mobile applications. *International Journal of Design, Analysis and Tools for Circuits and Systems*, 2(1) :14–29, August 2011.

[Kel06] Pierre Kelsen. A declarative executable model for object-based systems based on functional decomposition. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *ICSOFT (1)*, pages 63–71. INSTICC Press, 2006.

[KM08] Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 690–704, Berlin, Heidelberg, 2008. Springer-Verlag.

[KOHM14] Anssi Kostiainen, Ilkka Oksanen, and Dominique Hazaël-Massieux. Html media capture. w3c candidate recommandation. Technical report, W3C, September 2014.

[Kos12] Adrian Kosmaczewski. *Mobile JavaScript Application Development*. O'Reilly Media, June 2012.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3) :26–49, August 1988.

[Kra11] Frank Alexander Kraemer. Engineering android applications based on uml activities. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2011.

[LGA10] Lee Lundrigan, Vidal Graupera, and Sarah Allen. *Pro smartphone cross-platform development - iPhone, BlackBerry, Windows Mobile, and Android Development and Distriubution*. Apress, PA USA, 2010. Bibliothek FH Salzburg.

[LGW13] Olivier Le Goaer and Sacha Waltham. Yet another dsl for cross-platforms mobile development. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, pages 28–33, New York, NY, USA, 2013. ACM.

[LP06] L. Lefevre and J. Pierson. Just in time entertainment deployment on mobile platforms. In *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 117–117, Feb 2006.

[LT13] Lori Lalonde and David R. Totzke. *Windows Phone 8 Recipes : A Problem-Solution Approach*. Apress, Berkely, CA, USA, 1st edition, 2013.

[LY] Tim Lindholm and Frank Yellin. The java virtual machine specification. Technical report, Sun.

[Mar03] Robert C. Martin. The visitor family of design patterns. In *The Principles, Patterns, and Practices of Agile Software Development*. Prentice Hall PTR, 2003.

[McI68] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[MGL+11] Verdi March, Yan Gu, Erwin Leonardi, George Goh, Markus Kirchberg, and Bu Sung Lee. μcloud : Towards a new paradigm of rich mobile applications. *Procedia Computer Science*, 5(0) :618 – 624, 2011. The 2nd International Conference on Ambient Systems, Networks and Technologies (ANT-2011) / The 8th International Conference on Mobile Web Information Systems (MobiWIS 2011).

[MKS+11] Bup-Ki Min, Minhyuk Ko, Yongjin Seo, Seunghak Kuk, and Hyeon-Soo Kim. A uml metamodel for smart device application modeling based on windows phone 7 platform. In *TENCON 2011 - 2011 IEEE Region 10 Conference*, pages 201–205, Nov 2011.

[MMOR09] Patricia Miravet, Ignacio Marín, Francisco Ortin, and Abel Rionda. DIMAG : a framework for automatic generation of mobile applications for multiple platforms. In *Proceedings of the 6th International Conference on Mobile Technology, Applications, and Systems, Mobility Conference 2009, Nice, France, September 2-4, 2009*. ACM, 2009.

[MRT99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 44–53. ACM, 1999.

[MSG+13] Nikunj Mehta, Jonas Sicking, Eliot Graff, Andrei Popescu, Jeremy Orlow, and Joshua Bell. Indexed database api. w3c candidate recommendation. Technical report, W3C, July 2013.

[MT13] Tommi Mikkonen and Antero Taivalsaari. Cloud computing and its impact on mobile software development : Two roads diverged. *Journal of Systems and Software*, 86(9) :2318 – 2320, 2013.

[Nal11] A. Nalwaya. *Rhomobile Beginner's Guide : Step-by-step Instructions to Build an Enterprise Mobile Web Application from Scratch*. Learn by doing : less theory, more results. Packt Publishing, 2011.

[Neu13] Matt Neuburg. *iOS 7 Programming Fundamentals : Objective-C, Xcode, and Cocoa Basics*. O'Reilly Media, Inc., 2013.

[omg06] Corba component model (v4). Technical report, OMG, April 2006.

[omg14] Interface definition language (v3.5). Technical report, OMG, March 2014.

[oPDL14] oachim Perchat, Mikael Desertot, and Sylvain Lecomte. Common framework : A hybrid approach to integrate cross-platform components in mobile application. *Journal of Computer Science*, 2014.

[OSG14] OSGi. Osgi service platform (6th release),. June 2014.

[PA13] Arno Puder and Oren Antebi. Cross-compiling android applications to ios and windows phone 7. *MONET*, 18(1) :3–21, 2013.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PBJ98] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup : Architecture for component trading and dynamic updating. *Configurable Distributed Systems, International Conference on*, 0 :43, 1998.

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[PCBD10] Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. Feature-based Composition of Software Architectures. In Muhammad Ali Babar and Ian Gorton, editors, *4th European Conference on Software Architecture*, Lecture Notes in Computer Science 6285, pages 230–245, Copenhagen, Danemark, August 2010. Acceptance rate : 19/75 (25%).

[PD12] Arno Puder and Spoorthi D'Silva. Mapping objective-c API to java. In *Mobile Computing, Applications, and Services - 4th International Conference, MobiCASE 2012, Seattle, WA, USA, October 11-12, 2012. Revised Selected Papers*, pages 21–43, 2012.

[PDL12] J. Perchat, M. Desertot, and S. Lecomte. Cross-platform : le nouveau challenge dans le domaine du mobile. In Cépaduès, editor, *Ubimob'12*, 2012.

[PDL13] Joachim Perchat, Mikael Desertot, and Sylvain Lecomte. Component based framework to create mobile cross-platform applications. In Elhadi Shakshuki, Karim Djouani, Michael Sheng, Mohamed Younis, Eduardo Vaz, and Wayne Groszko, editors, *Proceedings of the 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013), the 3rd International Conference on Sustainable Energy Information Technology (SEIT-2013), Halifax, Nova Scotia, Canada, June 25-28, 2013*, volume 19 of *Procedia Computer Science*, pages 1004–1011. Elsevier, 2013.

[PDLP11] Dana Popovici, Mikael Desertot, Sylvain Lecomte, and Nicolas Peon. Context-aware transportation services (CATS) framework for mobile environments. *IJNGC*, 2(1), 2011.

[Per11] Joachim Perchat. Études des outils prenant en compte l'hétérogénéité des terminaux mobiles pour le déploiement à grande échelle de services. Rapport de stage, September 2011.

[Pop14] Andrei Popescu. Geolocation api specification. w3c working draft. Technical report, W3C, July 2014.

[PQD12] Carlos Andres Parra, Clément Quinton, and Laurence Duchien. Capucine : Context-aware service-oriented product line for mobile apps. *ERCIM News*, 2012(88), 2012.

[PSC12] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. Comparison of cross-platform mobile development tools. In *Procs. of the 16th Intl. Conf. on Intelligence in Next Generation Networks (ICIN 2012)*. IEEE Digital Library, October 2012.

[PTM14] Arno Puder, Nikolai Tillmann, and Michal Moskal. Exposing native device apis to web apps. In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems, MOBILESoft 2014, Hyderabad, India, June 2-3, 2014*, pages 18–26, 2014.

[Pud10] Arno Puder. Cross-compiling android applications to the iphone. In *Proceedings of the 8th International Conference on Principles and Practice of Programming in Java, PPPJ 2010, Vienna, Austria, September 15-17, 2010*, pages 69–77, 2010.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11) :1056–1076, 2002.

[PXL10] Biao Pan, Kun Xiao, and Lei Luo. Component-based mobile web application of cross-platform. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10. IEEE Computer Society, 2010.

[QDDD11] Clément Quinton, Christophe Demarey, Nicolas Dolet, and Laurence Duchien. AppliDE : modélisation et génération d'applications pour smartphones. In *Journées sur l'Ingénierie Dirigée par les Modèles (IDM'11)*, pages 41–45, Lille, France, June 2011.

[QMPD11] Clément Quinton, Sébastien Mosser, Carlos Parra, and Laurence Duchien. Using multiple feature models to design applications for mobile phones. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC '11, pages 23 :1–23 :8, New York, NY, USA, 2011. ACM.

[RB10] Andrew Lee Rubinger and Bill Burke. *Enterprise JavaBeans 3.1 - Developing Enterprise Java Components : Covers JavanBeans 3.1 (6. ed.)*. O'Reilly, 2010.

[RdS12] Andre Ribeiro and Alberto Rodrigues da Silva. Survey on cross-platforms and languages for mobile apps. *2012 Eighth International Conference on the Quality of Information and Communications Technology*, 0 :255–260, 2012.

[RdS14] André Ribeiro and Alberto Rodrigues da Silva. Xis-mobile : a DSL for mobile applications. In Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong, editors, *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1316–1323. ACM, 2014.

[REUKH13] Jussi Ronkainen, Juho Eskeli, Timo Urhemaa, and Kaisa Koskela-Huotari. Experiences on mobile cross-platform application development using phonegap. In *Proceedings of The Sixth International Conference on Advances in Human oriented and Personalized Mechanisms, Technologies, and Services*, 2013.

[Ris08] Ray Rischpater. *Beginning Java ME Platform*. Apress, Berkely, CA, USA, 1 edition, 2008.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[TVBP14] Rich Tibbett, Tim Volodine, Steve Block, and Andrei Popescu. Deviceorientation event specification. w3c working draft. Technical report, W3C, March 2014.

[Van05] Jean Vanderdonckt. A mda-compliant environment for developing user interfaces of information systems. In *CAiSE*, pages 16–31, 2005.

[vis13] Developer economics q3 2013 : State of the developer nation. Technical report, VisionMobile, July 2013.

[War14] John M. Wargo. *Apache Cordova API Cookbook*. July 2014.

[Was10] Anthony I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM.

[WLL10] Yonghong Wu, Jianchao Luo, and Lei Luo. Porting mobile web application engine to the android platform. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 2157–2161, Washington, DC, USA, 2010. IEEE Computer Society.

[xam14] Introduction to mobile development. understanding xamarin mobile application projects. Technical report, Xamarin, 2014.

[Yua03] Michael Juntao Yuan. *Enterprise J2ME : Developing Mobile Java Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[ZJKG10] Xinwen Zhang, Sangoh Jeong, Anugeetha Kunjithapatham, and Simon Gibbs. Towards an elastic application model for augmenting computing capabilities of mobile platforms. In Ying Cai, Thomas Magedanz, Minglu Li, Jinchun Xia, and Carlo Giannelli, editors, *Mobile Wireless Middleware, Operating Systems,*

and Applications, volume 48 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 161–174. Springer Berlin Heidelberg, 2010.

[ZUV10] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In Muhammad Ali Babar and Ian Gorton, editors, *Software Architecture, 4th European Conference, ECSA 2010, Copenhagen, Denmark, August 23-26, 2010. Proceedings*, volume 6285 of *Lecture Notes in Computer Science*, pages 295–310. Springer, 2010.